VISIT…

Let Run your Neurons

team LRN

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

3D Game Programming All in One

Kenneth C. Finney

Includes CD-ROM

Series Editor
André LaMothe, CEO, Xtreme Games LLC

Premier Press

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

3D Game Programming All in One

Kenneth C. Finney

Series Editor
André LaMothe, CEO, Xtreme Games LLC

Team LRN

This page intentionally left blank

Team LRN

3D Game Programming All in One

Kenneth C. Finney

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

Team LRN

© 2004 by Premier Press, a division of Course Technology. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without written permission from Course PTR, except for the inclusion of brief quotations in a review.

The Premier Press logo and related trade dress are trademarks of Premier Press and may not be used without written permission.

UltraEdit is a registered trademark of IDM Computer Solutions, Inc., Paint Shop Pro 8 is a trademark of Jasc Corporation, Inc. Audacity and QuArK 6.3 use are subject to the GNU General Public License. Chain Reaction and Reaction Engine SDK are trademarks of Monster Studios. UVMapper 0.25—copyright ©1998-2002 Stephen L Cox, All rights reserved. ThinkTanks is a trademark of BraveTree Productions, LLC. Orbz is a trademark of Mind Vision Software. Marble Blast Gold is a trademark of GarageGames. MilkShape 3D is a trademark of chUmbaLum sOft.

All other trademarks are the property of their respective owners.

Important: Course PTR cannot provide software support. Please contact the appropriate software manufacturer's technical support line or Web site for assistance.

Course PTR and the author have attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Information contained in this book has been obtained by Course PTR from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Course PTR, or others, the Publisher does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information. Readers should be particularly aware of the fact that the Internet is an ever-changing entity. Some facts may have changed since this book went to press.

Educational facilities, companies, and organizations interested in multiple copies or licensing of this book should contact the publisher for quantity discount information. Training manuals, CD-ROMs, and portions of this book are also available individually or can be tailored for specific needs.

ISBN: 1-59200-136-X

Library of Congress Catalog Card Number: 2004090733

Printed in the United States of America

04 05 06 07 08 BH 10 9 8 7 6 5 4 3 2 1

THOMSON

COURSE TECHNOLOGY

Professional ■ Trade ■ Reference

Course PTR, a division of Course Technology
25 Thomson Place
Boston, MA 02210
http://www.courseptr.com

SVP, Course Professional, Trade, Reference Group:
Andy Shafran

Publisher:
Stacy L. Hiquet

Senior Marketing Manager:
Sarah O'Donnell

Marketing Manager:
Heather Hurley

Manager of Editorial Services:
Heather Talbot

Acquisitions Editor:
Mitzi Koontz

Associate Marketing Manager:
Kristin Eisenzopf

Series Editor:
André LaMothe

Developmental Editors:
Dave Astle and Kevin Hawkins

Project Editor:
Jenny Davidson

Technical Reviewers:
Michael Dawson and Les Pardew

Retail Market Coordinator:
Sarah Dubois

Copy Editor:
Laura Gabler

Interior Layout Tech:
Jill Flores

Cover Designer:
Steve Deschene

CD-ROM Producer:
Brandon Penticuff

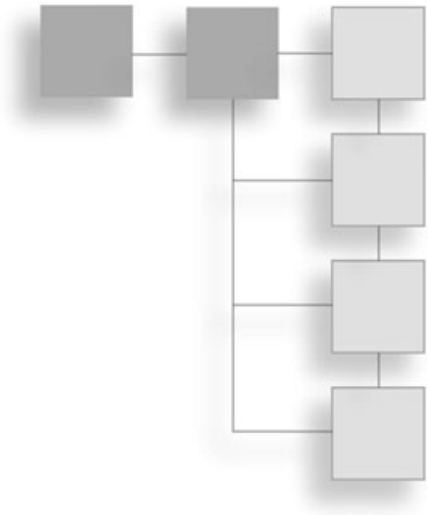
Indexer:
Sharon Shock

Proofreaders:
Sandi Wilson and Sara Gullion

Team LRN

This book is dedicated to my wife, Tubetti, and my two sons, Rockid and Lincus, whose sacrifices and encouragement made it possible.

-CERDIP



Acknowledgments

I would like to thank Dave Wilkes for his encouragement to do this book, and the other guys at Wilkes Associates for just putting up with me, especially during the early days of its creation.

I also want to thank my editors, Mitzi Koontz, Laura Gabler, Mike Dawson, Les Pardew, Kevin Hawkins, and Dave Astle, and especially the ever-patient Jenny Davidson (she laughs at my jokes!). A big thank you to André LaMothe for pushing the idea, and making it happen.

Many thanks and a tip o' the hat go to those Four Guys in a Garage: Jeff Tunnell, Rick Overman, Mark Frohnmayer, and Tim Gift. These are the perpetrators of Torque, and the founders of GarageGames. An amazing crew. Thanks to Desmond Fletcher for his assistance (knowing and unknowing) with subjects as diverse as particles, terrain, and clouds. Many thanks go to Melv May, Harold Brown, Anthony Rosenbaum, Phil Carlisle, Dave Wyand, Matthew Fairfax, Pat Wilson, Ryan Parker, Simon Windmill, Kevin Ryan, Joe Maruschak, Joel Baxter, Justin Mette and the 21-6 gang, and Frank Bignone, for their many contributions to the Torque engine and its game development community. Hearty thanks to Nick Palmer for allowing me to use his music, which appears on the CD.

I also want to thank every player who came to Tubettiworld in those halcyon DF2 days and made it his or her virtual home. They made it a great place to play and socialize online. I would like to list them all, but obviously I can't. To the late John "Tufat" Tucker, the gentleman—I salute you, !S. Then there are, in no particular order: AceTW, his evil twin Malfunction, Strata, Spector, Roadkill, Midnight, Oz Mal, Deadbolt, Insomniac, Checkfire, Norway, Animal, Qdad, MickyD, Buster, Major Chip Hazard, Pirate, Kotch, C2, FF6, IRS Agent, and Kdawg—I mustn't neglect to mention Dr. Evil and the great work he

Team LRN

and his gang are doing with the TXP stuff. Last, but certainly not least, Jim, The Nailer, the epitome of the Online Game Player, and an all around great guy. I hope that everything works out, Jim.

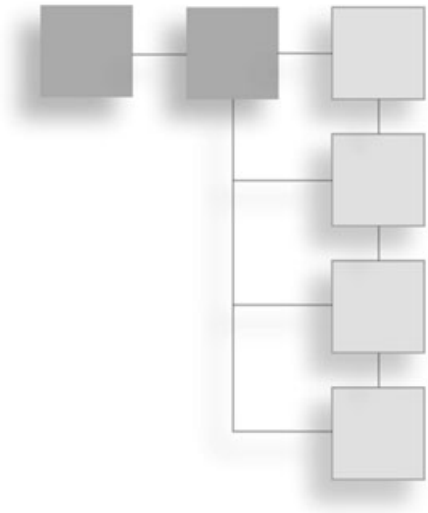
Along the way, there have been many others in various places that deserve some mention: KILLER and his gang, who do what cornered rats do best—fight back. Many other game developers can learn a thing or two about hard work from those guys. Onchas, Cowboy, Badger, and the rest of the "Allies"—keep up the good work. Same with you "Axis" players (except that your days are numbered!). Also a hearty !S to the Playnet forum denizens who opened a second front as soon as the war started (Teh?).

I'm sure I've forgotten to acknowledge someone, and I'll probably get e-mails to that effect, but that's the risk one embraces.

Regards,

CERDIP

Team LRN



About the Author

Kenneth C. Finney is the Principal Software Engineer at Wilkes Associates, Inc. in the Greater Toronto Area. He began programming in 1974 and was a recipient of the prestigious Conference Board of Canada ITX (Innovation in Technology Excellence) Award in 1997 for his work on InScan—a high-speed document scanning system. He was a consultant to the Department of National Defence in Canada in Armoured Fighting Vehicle systems design, and is an orange-qualified Nuclear Engineer designing NDE systems and techniques for Candu reactor stations. He is an associate professor at Seneca College at York University in Toronto, helping technical writers learn how to survive in a software development environment. Ken is the creator of the popular Tubettiland 'Online Campaign' Mod and the 'QuicknDirty' game management tools for Novalogic's Delta Force 2 game series. He is currently working on the new and unique Tubettiworld action/adventure game (www.tubettiworld.com) using the Torque Game Engine.

Team LRN

About the Series Editor

André LaMothe, CEO, Xtreme Games LLC, has been involved in the computing industry for more than 25 years. He wrote his first game for the TRS-80 and has been hooked ever since! His experience includes 2D/3D graphics, AI research at NASA, compiler design, robotics, virtual reality, and telecommunications. His books are top sellers in the game programming genre, and his experience is echoed in the Course Technology PTR *Game Development* series.

Team LRN

Contents at a Glance

Introduction xxvi

Chapter 1
Introduction to 3D Game Development 1

Chapter 2
Introduction to Programming 31

Chapter 3
3D Programming Concepts 89

Chapter 4
Game Programming 123

Chapter 5
Game Play 157

Chapter 6
Network 205

Chapter 7
Common Scripts 235

Chapter 8
Introduction to Textures 275

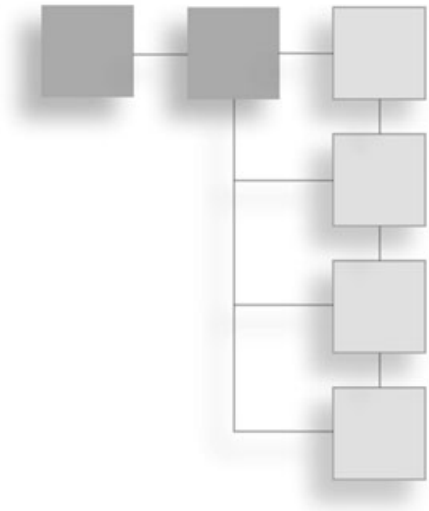
Chapter 9
Skins 309

Chapter 10
Creating GUI Elements 335

Chapter 11
Structural Material Textures 351

Team LRN

Chapter 12
Terrains 365
Chapter 13
Introduction to Modeling with MilkShape 381
Chapter 14
Making a Character Model 415
Chapter 15
Making a Vehicle Model 465
Chapter 16
Making Weapons and Items 479
Chapter 17
Making Structures 499
Chapter 18
Making the Game World Environment 513
Chapter 19
Creating and Programming Sound 539
Chapter 20
Game Sound and Music 559
Chapter 21
Creating the Game Mission 583
Chapter 22
The Game Server 609
Chapter 23
The Game Client 631
Chapter 24
The End Game 659
Appendix A
The Torque Game Engine Reference 667
Appendix B
Game Development Resources on the Internet 741
Appendix C
Game Development Tool Reference 749
Appendix D
QuArK Reference 755
Index 773



Contents

Introduction xxvi

Chapter 1 Introduction to 3D Game Development 1

The Computer Game Industry 1
- 3D Game Genres and Styles 2
- Game Platforms 8
- Game Developer Roles 11
- Publishing Your Game 15

Elements of a 3D Game 16
- Game Engine 16
- Scripts 17
- Graphical User Interface 19
- Models 19
- Textures 20
- Sound 20
- Music 21
- Support Infrastructure 21

The Torque Game Engine 23
- Descriptions 23
- Using Torque in This Book 28
- Moving Right Along 29

Team LRN

Chapter 2 Introduction to Programming 31
UltraEdit-32 31
Program Setup and Configuration 32
Setting Up Projects and Files 32
Search and Replace 35
Find in Files 38
grep 39
Bookmarks 42
Macros 43
UltraEdit Review 44
Controlling Computers with Programs 45
Programming Concepts 48
How to Create and Run the Example Programs 48
Hello World 49
Expressions 52
Variables 53
Operators 60
Loops 64
Functions 66
Conditional Expressions 71
Branching 74
Debugging and Problem Solving 82
Best Practices 86
Moving Right Along 87

Chapter 3 3D Programming Concepts 89
3D Concepts 89
Coordinate Systems 90
3D Models 92
3D Shapes 94
Displaying 3D Models 95
Transformation 95
Rendering 98
Scene Graphs 103
3D Audio 104
3D Programming 104
Programmed Translation 105
Programmed Rotation 111

Team LRN

Programmed Scaling 113
Programmed Animation 115
3D Audio 119
Moving Right Along 122

Chapter 4 Game Programming 123
Torque Script 123
Strings 124
Objects 125
DataBlocks 128
Game Structure 129
Server versus Client Design Issues 132
Common Functionality 133
Preparation 133
Root Main 134
Control Main 139
Initialization 141
Client 144
Server 149
Player 151
Running Emaga4 153
Moving Right Along 155

Chapter 5 Game Play 157
The Changes 157
Folders 157
Modules 158
Control Modules 158
control/main.cs 159
Client Control Modules 160
control/client/client.cs 160
control/client/interfaces/menuscreen.gui 162
control/client/interfaces/playerinterface.gui 165
control/client/interfaces/splashscreen.gui 169
control/client/misc/screens.cs 169
control/client/misc/presetkeys.cs 171
Server Control Modules 175
control/server/server.cs 175
control/server/players/player.cs 180

control/server/weapons/weapon.cs 186
control/server/weapons/crossbow.cs 190
control/server/misc/item.cs 197
Running Emaga5 202
Moving Right Along 203

Chapter 6 Network 205
Direct Messaging 205
CommandToServer 206
CommandToClient 207
Direct Messaging Wrap-up 209
Triggers 209
Area Triggers 209
Animation Triggers 209
Weapon State Triggers 210
Player Event Control Triggers 210
GameConnection Messages 211
What GameConnection Messages Do 212
Specifics 212
Finding Servers 217
Code Changes 217
New Modules 218
Dedicated Server 230
Root Main Module 230
Control—Main Module 231
Control—Initialize Module 231
Testing Emaga6 232
Moving Right Along 233

Chapter 7 Common Scripts 235
Game Initialization 235
Selected Common Server Modules 240
The Server Module 240
The Message Module 241
The MissionLoad Module 242
The MissionDownload Module 246
The ClientConnection Module 250
The Game Module 256

Selected Common Code Client Modules 258
The Canvas Module 259
The Mission Module 261
The MissionDownload Module 262
The Messages Module 266
A Final Word 268
Moving Right Along 273

Chapter 8 Introduction to Textures 275
Using Textures 275
Paint Shop Pro 279
Installing Paint Shop Pro 279
Getting Started 279
Working with Files 283
Paint Shop Pro Features 290
Moving Right Along 307

Chapter 9 Skins 309
UV Unwrapping 309
The Skin Creation Process 310
Making a Soup Can Skin 311
The Soup Can Skinning Procedure 311
Testing the Soup Can Skin 315
Making a Vehicle Skin 316
The Dune Buggy Diversion 316
Testing the Runabout Skin 321
Making a Player Skin 322
The Head and Neck 322
Hair and Hands 327
The Clothes 329
Trying It on for Size 333
Moving Right Along 333

Chapter 10 Creating GUI Elements 335
Controls 336
GuiChunkedBitmapCtrl 337
GuiControl 339
GuiTextCtrl 339
GuiButtonCtrl 340

GuiCheckBoxCtrl 341
GuiScrollCtrl 342
GuiTextListCtrl 343
GuiTextEditCtrl 344
The Torque GUI Editor 345
The Cook's Tour of the Editor 345
Moving Right Along 349

Chapter 11 Structural Material Textures 351
Sources 352
Photography 352
Original Artwork 357
Scaling Issues 358
Tiling 359
Texture Types 360
Irregular 360
Rough 361
Pebbled 361
Woodgrain 361
Smooth 361
Patterned 362
Fabric 362
Metallic 362
Reflective 362
Plastic 362
Moving Right Along 363

Chapter 12 Terrains 365
Terrains Explained 365
Terrain Characteristics 365
Terrain Data 367
Terrain Modeling 367
Height Maps 368
Terrain Cover 369
Tiling 369
Creating Terrains 370
The Height-Map Method 370
Applying Terrain Cover 378
Moving Right Along 380

Team LRN

Chapter 13 Introduction to Modeling with MilkShape 381
MilkShape 3D 381
Installing MilkShape 3D 381
The MilkShape 3D GUI 382
Navigating in Views 383
View Scale and Orientation 383
The Soup Can Revisited 384
Menus 391
The Toolbox 398
The Preferences Dialog Box 404
UVMapper 406
The File Menu 407
The Edit Menu 407
The Help Menu 407
UV Mapping 407
Moving Right Along 414

Chapter 14 Making a Character Model 415
Modeling Techniques 415
Shape Primitives 415
Incremental Polygon Construction 415
Axial Extrusion 416
Arbitrary Extrusion 417
Topographical Shape Mapping 417
Hybrids 417
The Base Hero Model 417
The Head 418
The Torso 423
Matching the Head to the Torso 429
The Legs 430
Integrating the Legs to the Torso 432
The Arms 433
Integrating the Arms to the Torso 438
The Hero Skin 438
Character Animation 443
Animating Characters in Torque 443
Building the Skeleton 446
Rigging: Attaching the Skeleton 447

Team LRN

Exporting the Model for Torque 458
The Torque DTS Exporter for MilkShape 459
The Torque Game Engine (DTS) Exporter Dialog Box 459
Special Materials 460
Animation Sequences 463
Moving Right Along 464

Chapter 15 Making a Vehicle Model 465
The Vehicle Model 466
The Sketch 466
The Model 467
The Wheels 476
Testing Your Runabout 477
Moving Right Along 478

Chapter 16 Making Weapons and Items 479
The Health Kit 479
The Model 479
Testing the Health Kit 480
A Rock 481
Testing the Rock 483
Trees 483
The Solid Tree 485
Testing the Solid Tree 487
The Billboard Tree 488
Testing the Billboard Tree 489
The Tommy Gun 490
Making the Model 490
Skinning the Tommy Gun 494
Testing the Tommy Gun 495
The Tommy Gun Script 497
Moving Right Along 497

Chapter 17 Making Structures 499
Installing QuArK 500
Using the Installer 500
Configuration 500
Quick Start 501

Building Bridges 505
Building a House 508
Moving Right Along 512

Chapter 18 Making the Game World Environment 513
Skyboxes 513
Creating the Skybox Images 516
Adjusting for Perspective 518
The Sky Mission Object 519
Cloud Layers 521
Cloud Specifications 521
Cloud Textures 522
Fog 523
Storms 524
Setting Up Sound 524
Storm Materials 528
Lightning 529
Rain 531
A Perfect Storm 532
Water Blocks 533
Terraforming 534
Moving Right Along 538

Chapter 19 Creating and Programming Sound 539
Audacity 540
Installing Audacity 540
Using Audacity 540
Audacity Reference 542
OpenAL 550
Audio Profiles and Data Blocks 550
Audio Descriptions 551
Trying It Out 553
Koob 555
Moving Right Along 558

Chapter 20 Game Sound and Music 559
Player Sounds 559
Footsteps 560
Utterances 563

Weapon Sounds 565
Vehicle Sounds 572
Environmental Sounds 578
Interface Sounds 579
Music 580
Moving Right Along 582

Chapter 21 Creating the Game Mission 583
Game Design 583
Requirements 584
Constraints 585
Koob 586
Torque Mission Editor 587
World Editor 589
Terrain Editor 590
Terrain Terraform Editor 592
Terrain Texture Editor 592
Mission Area Editor 593
Building the World 594
Particles 594
The Terrain 605
Items and Structures 606
Moving Right Along 608

Chapter 22 The Game Server 609
The Player-Character 609
Player Spawning 609
Vehicle Mounting 611
The Model 611
Server Code 612
Vehicle 617
Model 617
Datablock 617
Triggering Events 620
Creating Triggers 620
Scoring 622
Moving Right Along 630

Chapter 23 The Game Client 631
Client Interfaces 632
MenuScreen Interface 632
SoloPlay Interface 634
Host Interface 635
FindServer Interface 635
ChatBox Interface 636
MessageBox Interface 640
Client Code 642
MenuScreen Interface Code 642
SoloPlay Interface Code 643
Host Interface Code 647
FindServer Interface Code 648
ChatBox Interface Code 650
MessageBox Interface Code 652
Game Cycling 655
Final Change 657
Moving Right Along 657

Chapter 24 The End Game 659
Testing 660
Basics 660
Regression 660
Play Testing 661
Test Harnesses 661
Hosted Servers 661
Dedicated Servers 662
FPS Game Ideas 662
Other Genres 663
Modifying and Extending Torque 664
Go For It 665

Appendix A The Torque Game Engine Reference 667
Torque Console Script Command Reference 667
Torque Reference Tables 727

Appendix B Game Development Resources on the Internet 741
Torque-Related Web Sites 741
Game Development Web Sites 743

Appendix C Game Development Tool Reference 749
Shareware and Freeware Tools 750
Modeling 750
Image Editing 751
Programming Editing 751
Audio Editing 752
Retail Tools 752
GNU General Public License 754

Appendix D QuArK Reference 755
The Map Editor 755
Configuration Utility 764
General 764
Map 766
map2dif Reference 767

Index 773

Letter from the Series Editor

In the past few years, game development has become a huge subject, covering so many areas of technology and expertise that learning all the various aspects of game development would be a huge undertaking that would easily take 5-10 years to master. One of my goals with the Premier *Game Development* series was to cover each and every area of game development in depth, in a highly technical manner. However, sometimes you just want to know "how" to do something; you're not really interested in every single detail. Along these lines, I experimented with a totally beginner book titled *Game Programming All in One*, in which the reader is assumed to know nothing about game development, not even how to program! The book you're holding is really a follow-up to that book, albeit on a slightly different path. Instead of teaching general game programming from the ground up, *3D Game Programming All in One* teaches you how to make 3D games—period.

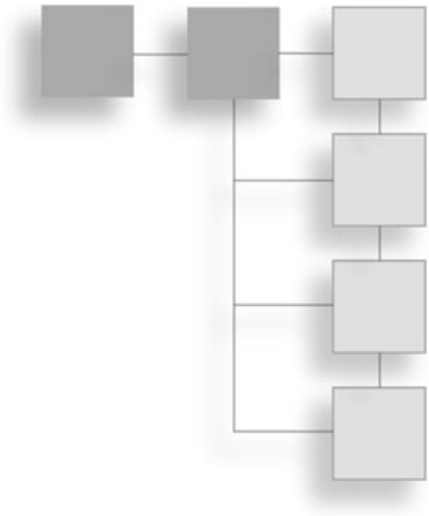
This book isn't so much about developing 3D engines, complex 3D math, or even physics, but how to create 3D games and what the high level major components of them are. As the author Kenneth Finney and I discussed and developed the book, we decided that the goal shouldn't be to exhaustively teach 3D game development—that would take 5,000 pages. Instead, the book should have the single goal—given a reader is familiar with C/C++, teach him how to make a 3D game as quickly as possible, leverage as much technology as possible, but still give the reader enough background information on the low-level aspects of 3D game development that if he did want to write everything from the rendered to the physics engine, he would have at least an idea of what they do.

3D Game Programming All in One is probably the only book you will find that will really live up to the hype of being able to teach you to create a 3D game. Let's face it, there are only a handful of people in the world that have the technical expertise (or the time) to write a commercial 3D engine, thus, this book saves you the time of that nightmare by leveraging one of the most advanced 3D engines available, the "Torque Engine"—even the name is cool. Ken uses this state-of-the-art 3D engine as a semi black box API to create 3D game examples in advancing levels of complexity. The book begins with basic 3D concepts, moves on to objects, models, large-scale worlds, and how all the elements of a 3D game fit together. Then Ken builds game demos that use these concepts, one added to another in a real-world example of developing an actual 3D game. By the end of the book, you will be able to create a number of 3D game types, from first-person shooters to exterior-based games with vehicles.

In conclusion, I highly recommend *3D Game Programming All in One* to anyone who wants to learn how to build 3D games, but doesn't necessarily want to spend 5-10 years learning how to build a 3D game engine from the ground up! In no time you will be creating amazing games based on a state-of-the-art engine. Then, if you so desire, you can always delve deeper into 3D engine design with further studies.

Sincerely,

André LaMothe
Series Editor, Course Technology PTR *Game Development* series



Introduction

Beginnings

> *"Hi, I'm using your software and I was wondering—can you tell me how I can make a computer game? I don't have much money, but I have this terrific idea for a shooter-like XYZ game, except I'll make it do…"*

During the past several years while working on the Tubettiland "Online Campaign" software and more recently while working on the *Tubettiworld* game, I figure I've received more than a hundred queries from people of all ages about how to get started making games. There were queries from 40-year-olds and 13-year-olds and every age in between. Most e-mails were from guys I would estimate to be in their late teens or early 20s.

After about the 30th response or so, I gave up trying to help these people out in detail and started to just point them to Web sites where they could gather the information they needed. Finally I stopped responding completely. But this bugged me to no end (I still get several of these e-mails in a month), so every now and then I will respond with the Web links or some pointers. However, whenever I do answer, I often get drawn into long e-mail exchanges for which I just don't have the time. Eventually I have to beg out of the exchange, usually by being nonresponsive at some point. Then I feel bad again.

I see this book as a sort of e-mail to everyone I haven't responded to. It's been rattling around in my head for about two years now, and I have to get it out!

About This Book

If you want to, you will be able to take this book and a computer, go into a room without Internet access, and emerge with a completed, ready-to-play first-person shooter game

Team LRN

within weeks. You will then be able to spend as much time as you want to dream up your game play concepts, and you will have the ability to add them to *your* game.

You might think this is a bold claim, but you can see for yourself. Go ahead and turn to the Table of Contents, or take a quick flip-through skim of the chapters. It's all there. If you follow through and do the exercises and work, you will arrive at the other end of the journey with experience, not just book learnin'.

Believe in Yourself

Computer games are a $9 billion per year industry. A growing part of this industry is people like you—part of a growing segment of the gamer population that doesn't just want to play the games but believes that you can *make* them better than the game companies can. Your problem may be that you lack the right combination of training, experience, and tools needed to turn your dreams into reality. This book is for you.

Every year more and more colleges offer game development programs, and every few months a new online indie game developer site launches on the Web. There is no lack of training available for those with the money to pay, and there is no lack of books for those of you who want to create your own engines or other specialized parts of a game.

The key element missing is a resource that takes the inspired and aspiring game developer by the hand and walks him through all the steps and tools required to make a fully featured game. This book is that resource. With the exception of game music composition (which itself could be a complete book series), you, the Gentle Reader, will *learn how to create every part of the game yourself* by using a well-defined toolkit of programs, knowledge, skills, and ideas. Sound, music, art, and code libraries are included on the companion CD for you to use if you lack a certain artistic or creative flair.

What You Bring to the Party

I assume that you have more than a passing familiarity with computer games, especially the first-person shooter genre. Throw in some computer savvy, add a reasonably capable computer system, sprinkle with desire, and you should be good to go!

Skills

You are probably fully able to deal with all aspects of Microsoft Windows–based computing. You don't need to be a programmer, but you do need to be aware that some programming will be required in creating a computer game. The first few chapters will introduce you to all the programming concepts that you will encounter in the course of using the book. You will not be expected to learn advanced 3D math in detail, but you will learn enough 3D math to accomplish your goals.

I'm going to show you how to create your own artwork, but you don't need to be an artist. The companion CD features a large collection of art you can use in your game.

System

All of the development tools, including the engine, are also included on the companion CD. All of these tools are priced such that even though the shareware version may be included on the CD, the actual registered versions are less than $100.

You will need a Windows-based computer to use this book. (The table below outlines the minimum system requirements.) It is possible for Macintosh and Linux users to use this book to create a game, because the game engine used—Torque—is also available for those platforms. However, not all of the required development tools are available on Mac and Linux, so the book's focus will be on Windows on Intel.

System Requirements

Processor	Pentium II/500MHz minimum
Operating System	Windows 98/ME/2000/XP
Video Card	3D graphics accelerated video card, NVidia GeForce 2–32MB equivalent or better
Display	17-inch recommended
Input Devices	keyboard and mouse
Memory	128MB minimum with 256MB recommended
Hard Disk	4GB minimum

What the Book Offers

In this book we are going to look at all aspects of game development, a journey from first principles to the completed game.

Concepts

We are going to take a look at various aspects of the game industry to give you the opportunity to see where you might fit in and what sort of opportunities there are. We'll also examine the elements of a 3D game, game design issues, and game genres.

Programming

Next, you'll be introduced to the programming concepts that you will need to understand in the course of using the book. You will see how to structure program code, create loops, call functions, and use globally and locally scoped variables. We'll use a subset of an object-oriented programming language called Torque Script, which is built into the

Torque Engine. Hands-on sample programs that you can try are available on the companion CD. We'll move on to examining the 3D concepts that you will need to understand some of the more sophisticated activities later in the book. This will provide a foundation for both the programming and the modeling tasks that you will take on later.

Torque

Once you've been powered up with sufficient knowledge and understanding of the main concepts in 3D game development, we'll get into using the Torque Engine in detail. You will learn how to handle client/server programming, how to control the player-character, how to send messages between players, how to create and control AI bots, and much more. Concepts will be presented with exercises and sample programs, which are available on the CD. Although we will cover some of the more intricate low-level workings of the Torque Engine in order to understand it better, it's important to realize that as an independent game developer you'll benefit more from mastering the higher-level functions that utilize the engine for us, so you can worry about other stuff—like game play. Without game play, you won't have a game.

Textures

Next, the book will show you everything you need to know about game textures: how to create them, how to modify and manipulate them, and how to use them in the game. The coverage is comprehensive; all of the texture types and their uses are discussed: skins, tiles, terrain, skyboxes, height maps, GUI widgets, and more. You will be guided through exercises in creating each of the texture types. A library of textures is available on the companion CD to fill in any gaps in your texture needs.

Models

Then we get to the meat of a 3D game—the models. In these chapters we will be delving into the world of low-poly modeling. We'll talk about the general principles involved in ways that can be applied to other tools, such as the expensive 3D MAX or Maya. But the practical focus will be geared toward using MilkShape, UVMapper, and other low-cost tools that are included on the companion CD.

I will show you the various model types, such as polygon-rendered or CSG models. You will create models for all aspects of the game in the exercises: player-characters, vehicles, weapons, powerups, decorations or clutter, buildings, and structures. You will walk through each step in the creation of the different model types so that you can create your own unique game look, if you want. All of the models in these chapters, plus many more, are available on the companion CD to round out your model library.

Sound and Music

After modeling, you will encounter the icing on the game cake: sound and music. You will discover how to select, create, and modify sounds for use in your game. You will also get some advice about selecting musical themes and how to integrate music into your game.

Integration

After picking up the required programming skills, and learning how to use the art creation and modeling tools, you will learn how to knit all the parts together to create a game, populate your game world, and then test and troubleshoot your game. Finally, we look at where you can go with your shiny new 3D game developer's toolkit of ideas, knowledge, skills, and software tools.

The Companion CD

The companion CD contains quite a few resources. Following is a quick description. For more detail, check the appendixes.

Source Code

The book's CD contains all of the Torque Script source code in sample form and final form. The samples will be aligned with the exercises in each chapter. The scripts for the final completed game will be included in its own directory tree. The game will be usable immediately upon installation from the CD so that you can have an instant and extensive preview of what is to come.

Game Engine

The CD will contain the complete Torque Game Engine with its executable, DLLs, and all required GUI and support files. It is a fully featured game engine that includes advanced networking capabilities, blended animations, built-in server-side anticheat capabilities, BSP support, a strong and complete object-oriented C++-like scripting language, and many other advanced features.

Tools

The following shareware tools are included on the CD:

- MilkShape 3D for 3D player and item modeling
- QuArK for 3D interior modeling
- Paint Shop Pro for texture and image manipulation
- Audacity for sound editing and recording
- UVMapper to perform UV unwrapping tasks
- UltraEdit-32 as the text or programming editor

Team LRN

Goodies

The CD also includes a few extras that aren't mentioned in the book or that are only briefly touched on:

- Retail games created with Torque: *Orbz, ThinkTanks, Marble Blast, Chain Reaction*
- Additional image and audio libraries
- Open Source utility source code

Go Get 'em!

The most important asset you have as an independent, and the key to any success, is your enthusiasm. Remember to use this book, and other books and training you acquire, as resources that will help you do what you want to do; they are not vouchers that you can trade in for a nice big pot of success. You have to do the work in the learning, and you have to do the work in the creating. And I know you can! Go get 'em!

Team LRN

This page intentionally left blank

Team LRN

CHAPTER 1

Introduction to 3D Game Development

Before we get into the nitty-gritty details of creating a game, we need to cover some background so that we can all work from the same page, so to speak. In the first part of this chapter, we will establish some common ground regarding the 3D game industry in the areas that matter—the types of games that are made and the different roles of the developers that make them. In the second part of the chapter, we'll establish what the essential elements of a 3D game are and how we will address them.

Throughout the book you will encounter references to different *genres*, or types, of games, usually mentioned as examples of where a particular feature is best suited or where a certain idea may have originally appeared. In this chapter we will discuss the most common of the 3D game genres. We will also discuss game development roles; I will lay out "job descriptions" for the roles of producer, designer, programmer, artist, and quality assurance specialist (or game tester). There are various views regarding the lines that divide the responsibilities so my descriptions are fairly generic.

Finally, we will discuss the concept of the 3D game engine. If ever there is going to be an area of dispute between a writer and his readers in a book like this, a discussion of what constitutes a 3D game engine will be it. I do have a trump card, though. In this book we will be using the Torque Game Engine as our model of what constitutes a fully featured 3D game engine. We will use its architecture as the framework for defining the internal divisions of labor of 3D game engines.

The Computer Game Industry

The computer game industry is somewhat different than other high-tech fields. The business operates more like Hollywood than traditional commercial or industrial software development; there are properties, producers, artists, and distributors. This industry has

Team LRN

its own celebrities. It is quite a bit more informal and relaxed than other high-tech fields in many ways but is quicker paced with a higher burnout rate. There are independent game developers, or indies, and big-name studios, but the computer game industry tends to be more entrepreneurial in spirit.

Just as in the motion picture industry, an indie developer is one that is not beholden to other businesses in their industry that can direct their efforts. Indies fund their own efforts, although they sometimes can get funding from outside sources, like a venture capitalist (good luck finding one). The key factor that makes them independent is that the funding does not come from *downstream* industry sources that would receive the developer's product, like a major game development house, publisher, or distributor.

Indies sell their product to distributors and publishers after the product is complete, or nearly so. If a developer creates a product under the direction of another company, they are no longer independent.

A good measure of the "indie-ness" of a developer is the answer to the following two questions:

- Can the developer make any game he wants, in whatever fashion he wants?
- Can the developer sell the game to whomever he wants?

If the answer is "yes" in both cases, then the developer is an indie.

Of course, another strong similarity with movies is that, as I pointed out earlier, games are typically classified as belonging to different genres.

3D Game Genres and Styles

Game development is a creative enterprise. There are ways to categorize the *game genres*, but I want you to keep in mind that while some games fit each genre like a glove, many others do not. That's the nature of creativity. Developers keep coming up with new ideas; sometimes they are jockeying for an advantage over the competition and sometimes they are just scratching an itch. At other times, calculating marketing departments decide that mixing two popular genres is a surefire path to a secure financial future.

The first rule of creative design is that there are no rules. If you are just scratching an itch, then more power to you. If you are looking to make a difference in the gaming world, you should at least understand the arena. Let's take a look at the most common 3D genres around today and a few that are interesting from a historical perspective. When you are trying to decide what sort of game you want to create, you should try understanding the genres and use them as guides to help focus your ideas.

It's important to note that all of the screen shots in this chapter are of games by indie game developers. Some of the games are currently being shipped as retail games, and

Team LRN

some are still in development. Almost all of them use the same Torque Game Engine we will use in this book to develop our own game.

By no means is this a definitive list; there are many genres that don't exist in the 3D gaming realm, and the number of ways of combining elements of genres is just too large to bother trying to enumerate. If you take pride in your creativity, you might resist attempts to pigeonhole your game idea into one of these genres, and I wouldn't blame you. When trying to communicate your ideas to others, however, you will find it useful to use the genres as shorthand for various collections of features, style, and game play.

Action Games

Action games come in several forms. The most popular are the *First-Person Point-of-View* (1st PPOV) games, where your player-character is armed, as are your opponents. The game play is executed through the eyes of your character. These sorts of games are usually called *First-Person Shooter* (FPS) games. Game play variations include *Death Match*, *Capture the Flag*, *Attack & Defend*, and *King-of-the-Hill*. Action games often have multiplayer online play, where your opponents are enemies controlled by real people instead of by a computer. Success in FPS games requires quick reflexes, good eye-hand coordination, and an intimate knowledge of the capabilities of your in-game weapons. Online FPS games are so popular that some games have no single-player game modes.

Some action games are strictly 3rd PPOV, where you view your player-character, or *avatar*, while also viewing the rest of the *virtual world* your avatar inhabits (see Figure 1.1).

Half-Life 2, *Rainbow Six*, and *Delta Force: Blackhawk Down* are popular examples of FPS-style action games.

Adventure Games

Adventure games are basically about exploring, where player-characters go on a quest, find things, and solve puzzles. The pioneering adventure games were text based. You would type in movement commands, and as you entered each new area or room, you would be given a brief description of where you were. Phrases like "You are in a maze of twisty passages, all alike" are now gaming classics. The best

Figure 1.1 *Think Tanks*—a 3rd PPOV action game made by BraveTree Productions using the Torque Game Engine.

adventure games play like interactive books or stories, where you as the player decide what happens next, to a certain degree.

Text adventures evolved into text-based games with static images giving the player a better idea of his surroundings. Eventually these merged with 3D modeling technology. The player was then presented with either a first- or third-person point of view of the scene his character was experiencing.

Adventure games are heavily story based and typically very linear. You have to find your way from one major accomplishment to the next. As the story develops, you soon become more capable of predicting where the game is going. Your success derives from your ability to anticipate and make the best choices.

Some well-known examples of adventure games are *The King's Quest* series, *The Longest Journey*, and *Syberia*.

Online adventure games have not really come into their own yet, although some games are emerging that might fit the genre. They tend to include elements of FPS action games and *Role-Playing Games* (RPGs) to fill out the game play, because the story aspect of the game is more difficult to accomplish in an online environment. Players advance at different speeds, so a monolithic linear story line would become pretty dreary to a more advanced player. An example of an online action-adventure-FPS hybrid game is *Tubettiworld* (see Figure 1.2), being developed by my all-volunteer team at Tubetti Enterprises.

Role-Playing Games

Role-playing games are very popular; that popularity can probably find its roots in our early childhood. At younger than age six or seven, we often imagined and acted out exciting adventures inspired by our action figures and other toys or children's books. As was also true for strategy games, the more mature forms of these games first evolved as pen-and-paper games, such as *Dungeons & Dragons*.

Figure 1.2 *Tubettiworld*—an action-adventure FPS hybrid game being developed by Tubetti Enterprises using the Torque Game Engine.

These games moved into the computer realm with the computer taking on more of the data-manipulation tasks of the game masters. In role-playing games, the player is usually

responsible for the development of his game character's skills, physical appearance, loyalties, and other characteristics. Eventually the game environment moved from each player's imaginations onto the computer, with rich 3D fantasy worlds populated by visually satisfying representations of buildings, monsters, and creatures (see Figure 1.3). RPGs are usually science fiction or fantasy based, with some historically oriented games being popular in certain niches.

Figure 1.3 *Myrmidon*—a science fiction RPG, another Torque-based game, being developed by 21-6 Productions.

Maze and Puzzle Games

Maze and puzzle games are somewhat similar to each other. In a maze game you need to find your way through a "physical" maze in which your routes are defined by walls and other barriers. Early maze games were 2D, viewed from the top; more recent ones play more like 3D adventure or FPS games.

Puzzle games are often like maze games but with problems that need to be solved, instead of physical barriers, to find your way through.

Mazes also make their appearance in arcade pinball-style games such as *Marble Blast* (see Figure 1.4) by GarageGames. It is a maze-and-puzzle hybrid game where you compete against the clock in an effort to navigate a marble around physical barriers. The puzzle aspect lies in determining the fastest (though not necessarily the most direct) route to the finish line.

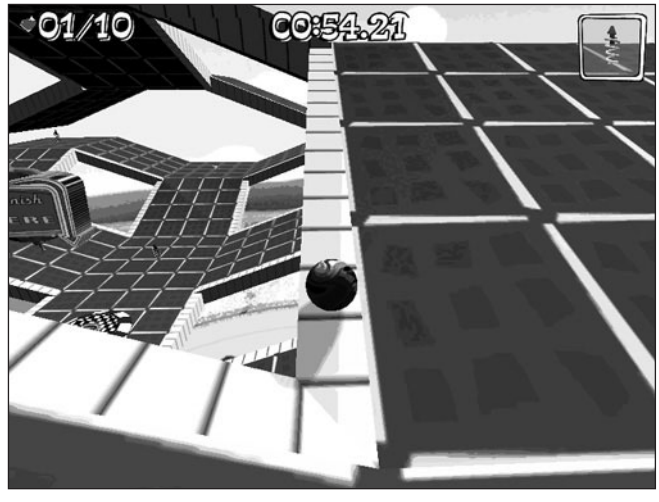


Figure 1.4 *Marble Blast*—a maze-and-puzzle hybrid game by GarageGames using its Torque Game Engine.

Puzzle games sometimes use puzzles that are variations of the shell game or that are more

indirect problem-solving puzzles where you must cause a series of things to happen in order to trigger some further action that lets you advance. Many puzzle games utilize direct problem-solving modes where the puzzle is presented visually. You then need to manipulate on-screen icons or controls in the correct sequences to solve the problem. The best puzzles are those where the solution can be deduced using logic. Puzzles that require pure trial-and-error problem-solving techniques tend to become tedious rather quickly. A historic example of a puzzle game is *The Incredible Machine* series by Dynamix. The latest variation of this type is the new game *Chain Reaction* by Monster Studios (see Figure 1.5).

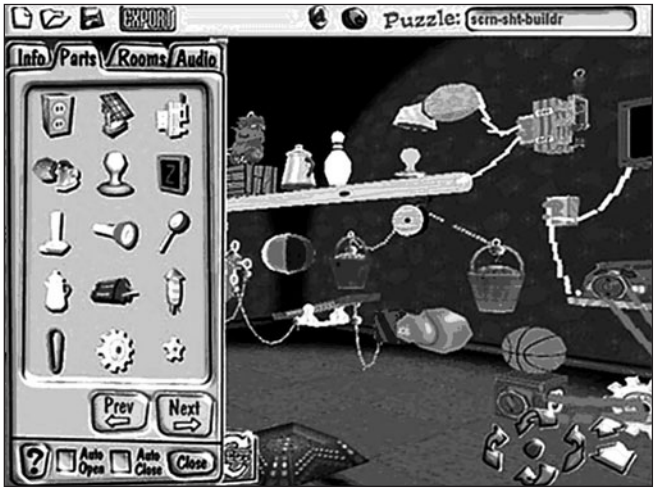


Figure 1.5 *Chain Reaction*—a puzzle game by Monster Studios using its Reaction Engine.

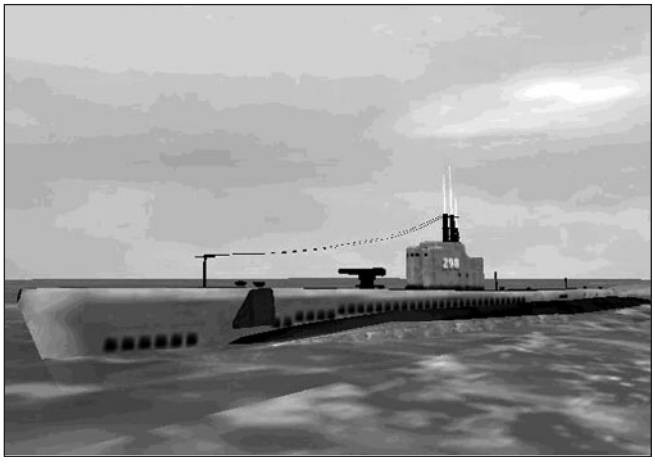


Figure 1.6 *Center World*—a submarine sim in development by Michael Hense, an independent game developer, using the Torque Game Engine.

Simulator Games

The goal of a simulator (or *sim*) game is to reproduce a real-world situation as accurately as possible. The measure of the simulation accuracy is usually called its *fidelity*. Most simulators put a heavy emphasis on the fidelity of the visual appearance, sounds, and physics of the game.

The point is total immersion in the game environment, so that you get the feeling you are actually there. You may be flying a jet fighter or driving a thoroughbred Grand Prix racing car. The game mirrors the real-life experience to the maximum the developers can manage.

Simulators usually require specialized input devices and controllers, such as aircraft joysticks and rudder pedals. Many simulator enthusiasts build complete physical cockpit mockups to enhance the immersion experience.

Falcon 4, *Grand Prix Legends*, and *Center World* (see Figure 1.6) are examples of simulator games.

Team LRN

Sports Games

Sports games are a variation of the simulator class of games in which the developer's intent is to reproduce the broad experience of the game as accurately as possible. You can participate in a sports game at various levels and watch the action play out in a realistic 3D environment (see Figure 1.7).

Figure 1.7 *Maximum Football*—a football sports game in development by David A. Winter, an independent game developer.

Unlike the action-oriented flight and driving simulators, sports games usually have a manager or season angle. While playing the game, you can also take on the role of coach, owner, or team manager. You can execute draft picks and trades or groom new players like any major league ball organization would. In a modern sports simulator you could be managing budgets, and you might play or race a regular year's schedule, playing in different stadiums or arenas or racing on different tracks.

Strategy Games

Strategy games began as pen-and-paper games, like war games, that have been around for centuries. As computer technology evolved, computer-based tables and random-number generators replaced the decision-making aspects of strategy games traditionally embodied by lookup charts and dice rolls.

Eventually the tabletop battlefields (or sandbox battlefields) with their cardboard markers or die-cast military miniatures moved into the computers as well. The early tabletop games were usually turn based: Each player would in turn consider his options and issue "orders" to his units. Then he would throw the dice to determine the result of the orders. The players would then modify the battlefield based upon the results. After this, the players would observe the new shape of the battlefield and plot their next moves. The cycle then repeated itself.

The advent of computer-based strategy games brought the concept of *real time* to the forefront. Now the computer determines the moves and results and then structures the battlefield accordingly. This has given birth to the Real-Time Strategy (RTS) genre. It does this on a time scale that reflects the action. Sometimes the computer will compress the time scale, and other times the computer will operate in real time, where one minute of

Team LRN

time in the game action takes one minute in the real world. The player issues orders to his unit as he deems them to be necessary. Recently, strategy games have moved into the 3D realm, where players can view the battlefield from different angles and perspectives as they plot their next moves (see Figure 1.8).

There are strategy games that exist outside the world of warfare. Some examples include business strategy games and political strategy games. Some of these games are evolving into *strategic simulations*, like the well-known *SimCity* series of games.

Game Platforms

This book is about computer games written for personal computers. There are three dominant operating systems: Microsoft Windows, Linux, and Mac OS. For some of these systems there are quite a few different flavors, but the differences within each system are usually negligible, or at least manageable.

Another obvious game platform type is the home game console, such as the Sony PlayStation or the Nintendo GameCube. These are indeed important, but because of the closed nature of the development tools and the expensive licenses required to create games for them, they are beyond the scope of this book.

Other game platforms include *Personal Digital Assistants* (PDAs), such as palm-based computers, and cell phones that support protocols that permit games to be played on them. Again, these platforms are also beyond the scope of this book.

Now that those little disclaimers are out of the way, let's take a closer look at the three game platforms of interest. It's important to note that by using the Torque Game Engine, you will be able to develop what amounts to a single code base for a game that you can ship for *all three platforms*: Windows, Linux, and Macintosh!

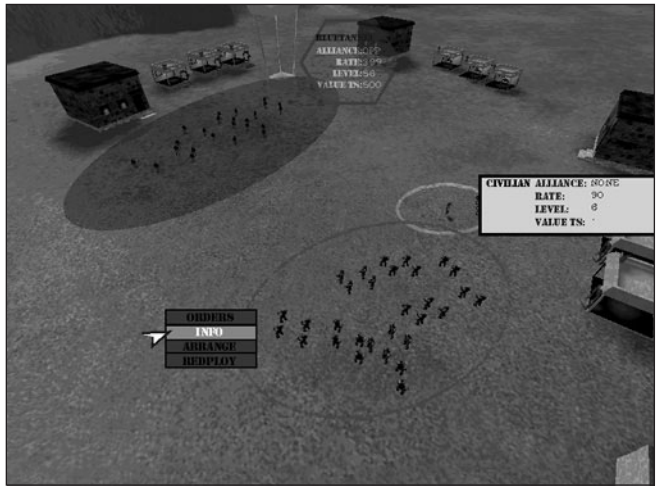


Figure 1.8 *Turf*—a 3D real-time multiplayer strategy game in development by Tubetti Enterprises, using a heavily modified version of the Torque Game Engine.

Windows

Windows has various historical versions, but the current flavors are Windows 2000, Windows XP, and the specialized Windows CE. In this book the expectation will be that you are developing on or for a

Some Popular Retail 3D Games and Their Genres

If you are still unclear about what a particular genre is about, take a look at the following table. It is a list of current "big-name" game titles (including one or two that are not yet released). Be aware that you may find a Web site or magazine somewhere that classifies these games in a slightly different way. That's cool—don't worry about it.

Game	Publisher	Genre
Age of Empires	Microsoft	Strategy
Battlefield 1942	Electronic Arts	Action-FPS
Civilization III	MicroProse	Strategy
Command & Conquer	Electronic Arts	RTS
Delta Force: Blackhawk Down	Novalogic	Action-FPS
Diablo	Blizzard	RPG
Doom III	Activision	Action-FPS
Duke Nukem Forever	Gathering of Developers	Action-FPS
Dungeon Siege	Microsoft	Action-RPG
Enter the Matrix	Infogrames	Action-FPS
Everquest	Sony	RPG
Grand Theft Auto: Vice City	Rockstar Games	Action-Sim
Half Life 2	Sierra	Action-FPS
Homeworld	Sierra	RTS
Medal of Honor: Allied Assault	Electronic Arts	Action-FPS
Myst III: Exile	UbiSoft	Adventure
PlanetSide	Sony	Action-FPS
Rainbow Six 3: Raven Shield	UbiSoft	Action-FPS
Return to Castle Wolfenstein	Activision	Action-FPS
SimCity 4	Electronic Arts	Strategy-Sim
Star Trek Elite Force 2	Activision	Action-FPS
Star Wars Jedi Knight 3	LucasArts	Action-FPS
Syberia	Microids	Adventure
The Longest Journey	Funcom	Adventure
Tom Clancy's Splinter Cell	UbiSoft	Action-FPS
Unreal II: The Awakening	Infogrames	Action-FPS
Unreal Tournament 2003	Infogrames	Action-FPS
WarCraft III: Reign of Chaos	Blizzard	RTS

Team LRN

Windows XP target system, because that is the version that Microsoft is now selling to the home computer market.

Within Windows XP, we will be using OpenGL and Direct3D as our low-level graphics *Application Programming Interfaces* (APIs). These APIs provide a means for our engine to access the features of the video adapters in our computers. Both OpenGL and Direct3D provide basically the same services, but each has its own strengths and weaknesses. With Torque you will have the choice of letting your end users use either API.

OpenGL's greatest strength lies in its availability with different computer systems. An obvious benefit is that the developer can create a game that will work on most computers. OpenGL is an open-source product. In a nutshell, this means that if there is a particular capability you want that OpenGL lacks, you can get access to the OpenGL source code and rebuild it the way you want. This assumes you have the skills, time, and tools necessary to get the job done, but you *can* do it.

DirectX is proprietary—it is the creation and intellectual property of Microsoft Corporation. Its biggest advantage is that it tends to support more features than OpenGL, and the 3D video adapter manufacturers tend to design their hardware to work with DirectX as much as they can. With DirectX you get a much more complete and the most advanced feature set. Unfortunately, you are limited to Windows-based systems if you put all your eggs in the DirectX basket.

The Torque Game Engine uses both APIs and gives you a rather straightforward set of techniques to set up your game with either. This means that in a Windows version of your game, you can offer your users the option of using the API that best suits their video adapter.

Linux

For most people, the single most important reason to use Linux is the price—it's free. You may have to pay to get a distribution of Linux on CD with manuals at a store, but you are paying for the cost of burning the CD, writing and printing the manuals, and distribution. You don't have to pay for the operating system itself. In fact, you can download Linux from many different locations on the Internet.

As a game developer, you will have a threefold interest in targeting Linux:

- Linux is a growing marketplace, and any market that is growing is a good target. Although the market is growing, it is still smaller than the Windows market. The place where Linux is growing is in universities, colleges, and other postsecondary institutions—and this is probably where your best computer gaming audience is.
- There are few computer games available for Linux desktops; most developers focus on Windows because it is the biggest market. If you ship a game for Linux, you will

Team LRN

be a bigger fish in a smaller ocean. That gets you exposure and a reputation that you can build on. And that's nothing to sneeze at.

- Linux offers a more configurable and secure environment for unattended Internet game servers. Linux servers can be run in a console mode that requires no fancy graphics, buttons, or mice. This allows you to utilize slower computers with less memory for servers and still get the computing power you need for your game server.

Unlike other operating systems, Linux comes in a variety of flavors known as *distributions*. There are many ongoing arguments about the merits of one distribution or another. Some of the more popular distributions are Red Hat, SuSE, Mandrake, Turbolinux, Debian, and Slackware. Although they may be organized differently in some cases and each has its own unique graphical look and feel, they are all based on the same kernel. It is the kernel that defines it as Linux.

Macintosh

The Macintosh is used a great deal in art-related fields and in the art departments of many businesses. Although the price point might not be as good as Linux (where the OS and most software is free), the Macintosh operating system is typically more accessible to the less tech-savvy users among us.

As with Linux, there has also traditionally been a dearth of computer games available for the Mac. So the big fish–small ocean factor applies here as well. Go ahead and make a splash!

note

One minor disadvantage of working with cross-platform software like Torque is the issue of naming conventions. In this book, wherever possible, I will head off the potential conflicts with a note that will cast a particular naming approach in stone for the duration of this book.

An example that will probably become obvious pretty quickly is the concept of *directories* or *folders*. The latter is shorter and easier to type, and the term will be used often. To save my editors the hassle, I will use *folders*. If you are a *directories* person, please just play along, okay?

Game Developer Roles

In the context of the game we will develop during our journey together through this book, you will wear all of the different game developer hats. The thing to remember is that oftentimes the lines between the roles will blur, and it might be hard to tell which hat you are wearing. So wear them all. Many indies wear multiple hats throughout the life of a game project, so it's just as well to get used to it!

Producer

A game producer is essentially the game project's leader. The producer will draw up and track the schedule, manage the people who do the hands-on development work, and manage the budget and expenditures. The producer may not know how to make any part of a game at all, but he is the one person on a game project who knows everything that is happening and why.

It's the producer who needs to poke the other developers in the ribs when they seem to be lagging. The producer needs to be aware when different members of the team are in need of some tool, knowledge, or resource and arrange to provide the team members with what they need.

Sometimes producers just need to spray a liberal dose of Ego-in-a-Can to refresh a despondent developer who keeps smashing into the same brick wall over and over while the clock ticks down.

The producer will also be the interface for the team to the rest of the world, handling media queries, negotiating contracts and licenses, and generally keeping the big noisy bothersome world off the backs of the development team.

Designer

If you are reading this, I have no doubt that you want to be a game designer. And why not? Game designers are like fun engineers—they create fun out of their imaginations. As a game designer, you will decide the theme and rules of the game, and you will guide the evolution of the overall feel of the game. And be warned—it had better be fun!

There are several levels of designers: lead designer, level designer, designer-writer, character designer, and so on. Large projects may have more than one person in each design role. Smaller projects may have only one designer or even a designer who also wears a programmer's or artist's hat! Or both!

Game designers need to be good communicators, and the best ones are great collaborators and persuaders. They need to get the ideas and concepts out of their heads and into the heads of the rest of the development team. Designers not only create the concept and feel of the game as a whole, but also create levels and maps and help the programmers stitch together different aspects of the game.

The lead designer will put together a design document that lays out all the aspects of the game. The rest of the team will work from this document as a guide for their work. A design document will include maps, sketches of game objects, plot devices, flow charts, and tables of characteristics. The designer will usually write a narrative text that describes how all of these parts fit together. A well-written and thorough game design completely describes the game from the player's perspective.

Team LRN

Unlike the producer, a designer needs to understand the technical aspects of the game and how the artists and programmers do what they do.

Programmer

Game programmers write program code that turns game ideas, artwork, sound, and music into a fully functional game. Game programmers control the speed and placement of the game artwork and sound. They control the cause-and-effect relationships of events, translating user inputs through internal calculations into visual and audio experiences.

There can be many different specializations in programming. In this book you will be doing a large amount of programming of game rules, character control, game event management, and scoring. You will be using Torque Script to do all these things.

For online game programming, specialization may also be divided between client code and server code. It is quite common to specify character and player behavior as a particular programmer specialty. Other specialty areas might be vehicle dynamics, environmental or weather control, and item management.

Other programmers on other projects might be creating parts of the 3D game engine, the networking code, the audio code, or tools for use with the engine. In our specific case these specializations aren't needed because Torque looks after all of these things for us. We are going to focus on making the game itself.

Visual Artist

During the design stages of development, game artists draw sketches and create storyboards to illustrate and flesh out the designers' concepts. Figure 1.9 demonstrates a conceptual design sketch created by a visual artist, and used by the development team as a reference for modeling and programming work. Later they will create all the models and texture artwork called for by the design document, including characters, buildings, vehicles, and icons.

The three principal types of 3D art are models, animations, and textures.

- 3D modelers design and build player-characters, creatures, vehicles, and other mobile 3D constructs. In order to ensure the game gets the best performance possible, model artists usually try to make the least complex model that suits the job. A 3D modeler is very much a sculptor working with digital clay.
- Animators make those models move. The same artist quite often does both modeling and animation.
- Texture artists create images that are wrapped around the constructs created by 3D modelers. Texture artists take photographs or paint pictures of various surfaces for use in these texture images. The texture is then wrapped around the objects in question in a process called *texture mapping*. Texture artists help the 3D modelers

reduce the model complexity by using highly detailed and cleverly designed textures. The intent is to fool the eye into seeing more detail than is actually there. If a 3D modeler molds a sculpture in digital clay, the texture artist paints that sculpture with digital paint.

Audio Artist

Audio artists compose the music and sound in a game. Good designers work with creative and inspired audio artists to create musical compositions that intensify the game experience.

Audio artists work closely with the game designers, determining where the sound effects are needed and what the character of the sounds should be. Audio artists often spend quite a bit of time experimenting with sound-effect sources, looking for different ways to generate the precise sound needed. Visit an audio artist at work and you might catch him slapping rulers and dropping boxes in front of a microphone. After capturing the basic sound, an audio artist will then massage the sound with sound-editing tools to vary the pitch, to speed it up or slow it down, to remove unwanted noise, and so on. It's often a

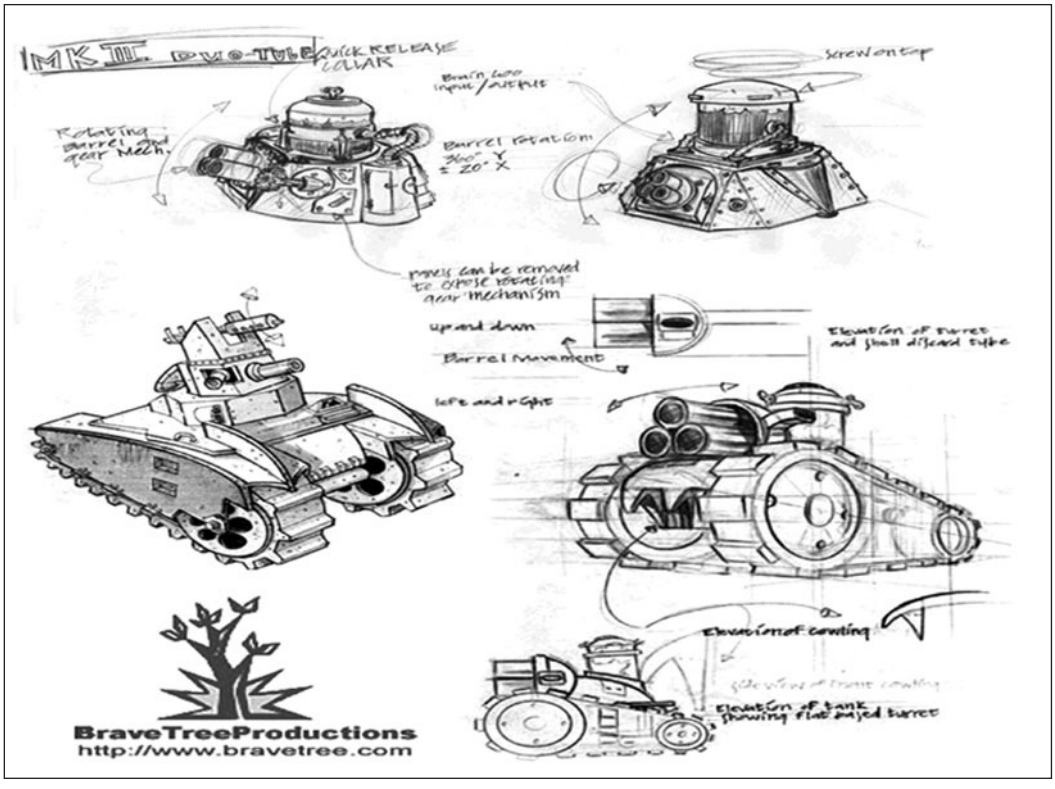


Figure 1.9 Conceptual design sketch.

tightrope walk, balancing realistic sounds with the need to exaggerate certain characteristics in order to make the right point in the game context.

Quality Assurance Specialist

Quality Assurance (QA) is a somewhat fancy name for *testing*. The general field of QA is more extensive than that, of course, but in the game business game testers take the brunt of the QA load. The purpose of testing is to ensure that a finished game is really finished, with as few bugs or problems as humanly possible. QA testing requires the quality assurance specialist, or game tester, to play each part of a game, trying to smooth out all glitches and bugs.

Most of the problems QA testing will find are visual or behavioral: text that doesn't correctly wrap on an edge, characters that don't jump correctly, or a level that has buildings misplaced. Testing can find game play problems; these are usually related more to the design than the programming. An example could be that the running speed of a player might not be fast enough to escape a particular enemy when it should be more than fast enough.

QA specialists need to be methodical in order to increase the chances of finding a bug. This might mean replaying a certain part of a game many times to the point of boredom. QA specialists need to be able to communicate well in order to write useful and meaningful bug reports.

Publishing Your Game

You can self-publish, of course. Whip up a Web site, add a shopping cart system, get your site added to various search engines, and sit back to wait for the dough to roll in, right? Well, it *might* work.

If you really think you have the next killer game and want it to sell, however, you need to hook up with someone who knows what they are doing. That would be a publisher. If you are an independent game developer, you will probably have difficulty attracting the attention of the big-name publishers. They usually know what they are looking for, are normally only interested in developers with proven track records, and probably already know whom they want to deal with anyway.

But all is not lost—there are options available for the indie. The one I recommend is GarageGames (http://www.garagegames.com). Besides offering competitive publishing terms for indie developers, GarageGames also created the Torque Game Engine, which it has graciously agreed to allow me to include on the CD for this book. Torque is the technology behind the popular and successful *Tribes* series of games. I'm going to help you learn how to use Torque as an enormous lever in creating your game.

But wait—there's more! If you really need to, you can buy a license from GarageGames for the Torque Game Engine that will give you (under the terms of the license) all of the

Team LRN

source code for the engine, so you can turn any game dream into a reality—for only $100! That's a hundred bucks for full access to the inner workings of an award-winning AAA 3D game engine. As Neo would say, "Whoa!"

I have no qualms about suggesting that you go to GarageGames. They are the guys behind the *Tribes* franchise, which is now owned by Sierra. They know their stuff, but they are not some big faceless corporate entity. They're basically a handful of guys who've made their splash in the corporate computer game industry, and now they're doing their level best to help the independent game developers of the world make their own splashes.

And no, they aren't paying for this book!

Elements of a 3D Game

The architecture of a modern 3D game encompasses several discrete elements: the engine, scripts, GUI, models, textures, audio, and support infrastructure. We're going to cover all of these elements in detail in this book. In this section I'll give you some brief sketches of each element to give you a sense of where we are going.

Game Engine

Game engines provide most of the significant features of a gaming environment: 3D scene rendering, networking, graphics, and scripting, to name a few. See Figure 1.10 for a block diagram that depicts the major feature areas.

Game engines also allow for a sophisticated rendering of game environments. Each game uses a different system to organize how the visual aspects of the game will be modeled. This becomes increasingly important as games are becoming more focused on 3D environments, rich textures and forms, and an overall realistic feel to the game. Textured Polygon rendering is one of the most common forms of rendering in FPS games, which tend to be some of the more visually immersive games on the market.

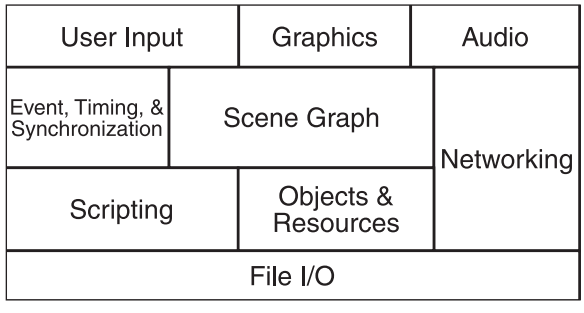


Figure 1.10 Elements of a game engine.

By creating consistent graphic environments and populating those environments with objects that obey specific physical laws and requirements, gaming engines allow games to progress significantly along the lines of producing more and more plausible narratives. Characters are constrained by rules that have realistic bases that increase the gamer's suspension of disbelief and draw him deeper into the game.

By including physics formulas, games are able to realistically account for moving bodies, falling objects, and particle movement. This is how FPS games such as *Tribes 2*, *Quake 3*, *Half-Life 2*, or *Unreal II* are able to allow characters to run, jump, and fall in a virtual game world. Game engines encapsulate real-world characteristics such as time, motion, the effects of gravity, and other natural physical laws. They provide the developer with the ability to almost directly interact with the gaming world created, leading to more immersive game environments.

As mentioned earlier, this book will employ the Torque Game Engine from GarageGames (http://www.garagegames.com). The Torque is included on the CD with this book. Later on we will discuss Torque in more detail—and you will understand why Torque was chosen.

Scripts

As you've just seen, the engine provides the code that does all the hard work, graphics rendering, networking, and so on. We tie all these capabilities together with scripts. Sophisticated and fully featured games can be difficult to create without scripting capability.

Scripts are used to bring the different parts of the engine together, provide the game play functions, and enable the game world rules. Some of the things we will do with scripts in this book include scoring, managing players, defining player and vehicle behaviors, and controlling GUI interfaces.

Following is an example of a Torque script code fragment:

```
// Beer::RechargeCompleteCB
// args: %this      - the current Beer object instance
//       %user      - the player connection user by id
//
// description:
//   Callback function invoked when the energy recharge
//   the player gets from drinking beer is finished.
//   Note: %this is not used.
function Beer:: RechargeCompleteCB (%this,%user)
{
       // fetch this player's regular recharge rate
       // and use it to restore his current recharge rate
       // back to normal
       %user.setRechargeRate(%user.getDataBlock().rechargeRate);
}

// Beer::OnUse
// args: %this      - the current Beer object instance
```

```
//       %user     - the player connection user by id
//
// description:
//   Callback function invoked when the energy recharge
//   the player gets from drinking beer is finished.
//
function Beer::OnUse(%this,%user)
{
   // if the player's current energy level
   // is zero, he can't be recharged, because
   // he is dying
   if (%user.getEnergyLevel() != 0)
   {
      // figure out how much the player imbibed
      // by tracking the portion of the beer used.
      %this.portionUsed += %this.portion;
      // check if we have used up all portions
      if (%this.portionUsed >= %this.portionCount)
      {
         // if portions used up, then remove this Beer from the
         // player's inventory and reset the portion
         %this.portionUsed = 0;
         %user.decInventory(%this,1);
      }
      // get the user's current recharge rate
      // and use it to set the temporary recharge rate
      %currentRate = %user.getRechargeRate();
      %user.setRechargeRate(%currentRate +%this.portionCount);

      // then schedule a callback to restore the recharge rate
      // back to normal in 5 seconds. Save the index into the schedule
      // list in the Beer object in case we need to cancel the
       // callback later before it gets called
      %this.staminaSchedule = %this.schedule(5000,"RechargeCompleteCB",%user);

      // if the user player hasn't just disconnected on us, and
      // is not a 'bot.
      if (%user.client)
      {
         // Play the 2D sound effect signifying relief ("ahhhhh")
         %user.client.play2D(Relief);
```

```
         // send the appropriate message to the client system message
         // window depending on whether the Beer has been finished,
         // or not. Note that whenever we get here portionUsed will be
         // non-zero as long as there is beer left in the tankard.
         if (%this.portionUsed == 0)
            messageClient(%user.client, 'MsgBeerUsed', '\c2Tankard polished off');
         else
            messageClient(%user.client, 'MsgBeerUsed', '\c2Beer swigged');
      }
   }
}
```

The example code establishes the rules for what happens when a player takes a drink of beer. Basically, it tracks how much of the beer has been consumed and gives the player a jolt of energy for five seconds after every mouthful. It sends messages to the player's client screen telling him what he's done—had a sip or polished off the whole thing. It also plays a sound effect of the player sighing in relief and contentment with every drink.

Graphical User Interface

The *Graphical User Interface* (GUI) is typically a combination of the graphics and the scripts that carries the visual appearance of the game and accepts the user's control inputs. The player's *Heads Up Display* (HUD), where health and score are displayed, is part of the GUI. So are the main start-up menus, the settings or option menus, the dialog boxes, and the various in-game message systems.

Figure 1.11 shows an example main screen using the *Tubettiworld* game. In the upper-left corner, the text that says "Client 1.62" is an example of a GUI text control. Stacked along the left side from the middle down are four GUI button controls. The popsicle-stick snapper logo in the lower right and the *Tubettiworld* logo across the top of the image are GUI bitmap controls that are overlayed on top of another GUI bitmap control (the background picture). Note that in the figure the top button control (Connect) is currently highlighted, with the mouse cursor over the top of it. This capability is provided by the Torque Game Engine as part of the definition of the button control.

In later chapters of this book we will spend a good deal of time contemplating, designing, and implementing the GUI elements of our game.

Models

3D models (Figure 1.12) are the essential soul of 3D games. With one or two exceptions, every visual item on a game screen that isn't part of the GUI is a model of some

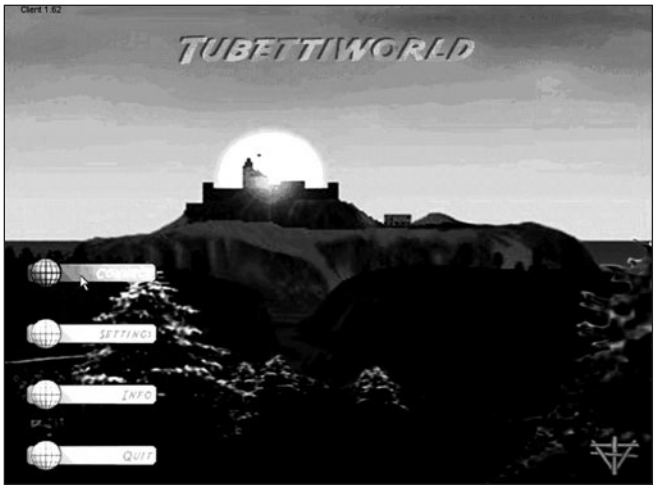


Figure 1.11 An example of a main menu GUI.

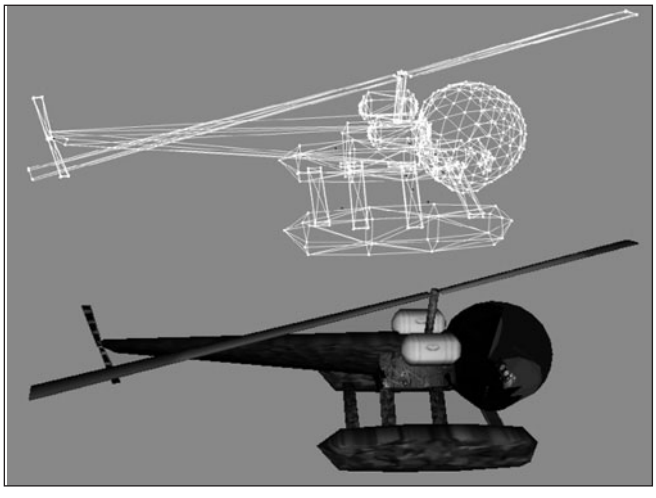


Figure 1.12 A 3D wire-frame and textured models of an old-style helicopter.

kind. Our player's character is a model. The world he tromps on is a special kind of model called *terrain*. All the buildings, trees, lampposts, and vehicles in our game world are models.

In later chapters we will spend a great deal of time creating and texturing models, animating them, and then inserting them into our game.

Textures

In a 3D game, textures are an important part of rendering the models in 3D scenes. Textures (in certain cases called *skins*—see Figure 1.13) define the visually rendered appearance of all those models that go into a 3D game. Proper and imaginative uses of textures on 3D models not only will enhance the model's appearance but will also help reduce the complexity of the model. This allows us to draw more models in a given period of time, enhancing performance.

Sound

Sound provides the contextual flavoring in a 3D game, providing audio cues to events and background sounds that imply environments and context, as well as 3D positioning cues for the player. Judicious use of appropriate sound effects is necessary for making a good 3D game. Figure 1.14 shows a sound-effect waveform being manipulated in a waveform-editing program.

Music

Some games, especially multiplayer games, use little music. For other games, such as single-player adventure games, music is an essential tool for establishing story line moods and contextual cues for the player.

Composing music for games is beyond the scope of this book. During the later chapters, however, I will point out places where music might be useful. It is always helpful to pay attention to your game play and whatever mood you are trying to achieve. Adding the right piece of music just might be what you need to achieve the desired mood.

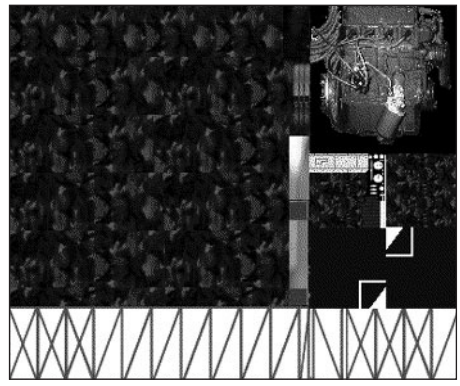


Figure 1.13 The textures used as the skin of the old-style helicopter.

Support Infrastructure

This is more important for persistent multiplayer online games than single player games. When we ponder game infrastructure issues, we are considering such things as databases for player scores and capabilities, auto-update tools, Web sites, support forums, and, finally, game administration and player management tools.

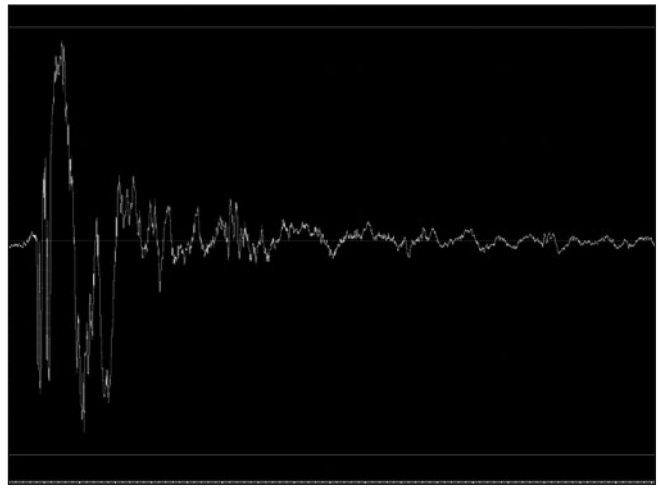


Figure 1.14 A graphical view of a gunshot sound-effect waveform.

The following infrastructure items are beyond the scope of this book, but I present them here to make you aware that you should spend time considering what you might need to do.

Web Sites

A Web site is necessary to provide people with a place to learn news about your game, find links to important or interesting information, and download patches and fixes for your game.

A Web site provides a focal point for your game, like a storefront. If you intend to sell your game, a well-designed Web site is a necessity.

Team LRN

Auto-update

An auto-update program accompanies your game onto the player's system. The updater is run at game start-up and connects via the Internet to a site that you specify, looking for updated files, patches, or other data that may have changed since the user last ran the program. It then downloads the appropriate files before launching the game using the updated information.

Games like *Delta Force: Blackhawk Down*, *World War II Online*, and *Everquest* have an auto-update feature. When you log in to the game, the server checks to see if you need to have any part of your installation upgraded, and if so, it automatically transfers the files to your client. Some auto-updaters will download a local installer program and run it on your machine to ensure that you have the latest files.

Support Forums

Community forums or bulletin boards are a valuable tool for the developer to provide to customers. Forums are a vibrant community where players can discuss your game, its features, and the matches or games they've played against each other. You can also use forums as a feedback mechanism for customer support.

Administrative Tools

If you are developing a persistent online game, it will be important to obtain Web-based tools for creating and deleting player accounts, changing passwords, and managing whatever other uses you might encounter. You will need some sort of hosted Web service with the ability to use CGI-, Perl-, or PHP-based interactive forms or pages. Although this is not strictly necessary, you really should invest in a database to accompany the administrative tools.

Database

If you intend your game to offer any sort of *persistence* where players' scores, accomplishments, and settings are saved—and need to be protected from fiddling by the players on their own computers—then you probably need a database back end. Typically, the administrative tools just mentioned are used to create player records in the database, and the game server communicates with the database to authenticate users, fetch and store scores, and save and recall game settings and configurations.

A common setup would include MySQL or PostgreSQL or something similar. Again, you will probably need to subscribe to a hosted Web service that offers a database.

The Torque Game Engine

I've mentioned the Torque Game Engine several times already. I think now would be a good time to take a little deeper look at the engine and how you will be using it.

Appendix A provides a reference for the Torque Game Engine, so look there if you really need more detail.

Descriptions

The following descriptions are by no means exhaustive, but a cup of coffee would go well with this section. Go ahead and make some—I'll wait. Black with two sweeteners, please.

Moving right along, you should note that the main reason for including this section is to give you, the Gentle Reader, the right sense of how much behind-the-scenes work is done for you by the engine.

Basic Control Flow

The Torque Game Engine initializes libraries and game functions and then cycles in the main game loop until the program is terminated. The main loop basically calls platform library functions to produce platform events, which then drive the main simulation.

Torque handles all of the basic event procession functions as follows:

- Dispatches Windows mouse movement events to the GUI
- Processes other input-related events
- Calculates elapsed time based on the time scale setting of the simulation
- Manages processing time for server objects
- Checks for server network packet transmissions
- Advances simulation event time
- Processes time for client objects
- Checks for client network packet transmission
- Renders the current frame
- Checks for network time-outs

Platform Layer

The platform layer provides a cross-platform architecture interface to the engine. The platform layer is responsible for handling file and network operations, graphics initialization, user input, and events.

Console

The console library provides the foundation for Torque-based games. The console has both a compiler and an interpreter. All GUIs, game objects, game logic, and interfaces are handled through the console. The console language is called Torque Script and is similar to a typeless C++, with some additional features that facilitate game development. You can load console scripts using a command from the console window as well as automatically from files.

Input Model

Input events are translated in the platform layer and then posted to the game. By default the game checks the input event against a global action map that supersedes all other action handlers. If there is no action specified for the event, it is passed on to the GUI system. If the GUI does not handle the input event, it is passed to the currently active (non-global) action map stack.

Platform-specific code translates Win32, Xwindows, or Mac events into uniform Torque input events. These events are posted into the main application event queue.

Action maps translate platform input events to console commands. Any platform input event can be bound in a single generic way—so in theory, the game doesn't need to know if the event came from the keyboard, the mouse, the joystick, or some other input device. This allows users of the game to map keys and actions according to their own preferences.

Simulation

A stream of events drives the game from the platform library: `InputEvent`, `MouseMoveEvent`, `PacketReceive-Event`, `TimeEvent`, `QuitEvent`, `ConsoleEvent`, `ConnectedReceive-Event`, `ConnectedAcceptEvent`, and `ConnectedNotifyEvent`. By journaling the stream of events from the platform layer, the game portion of the simulation session can be deterministically replayed for debugging purposes.

The simulation of objects is handled almost entirely in the game portion of the engine. Objects that need to be notified of the passage of time can be added to one of the two process lists: the global server or global client process list, depending on whether the object is a server object or a client ghost.

Server-side objects are only simulated at certain times, but client objects, in order to present a smooth view when the frame rate is high, are simulated after each time event.

There is a simulator class that manages all of the objects and events in the simulation. Objects are collected in a hierarchy of simulator classes and can be searched for by name or by object id.

Team LRN

Resource Manager

The Torque Engine uses many game resources. Terrain files, bitmaps, shapes, material lists, fonts, and interiors are all examples of game resources. Torque has a resource manager that it uses to manage large numbers of game resources and to provide a common interface for loading and saving resources. Under the auspices of Torque's resource manager, only one instance of a resource will ever be loaded at a time.

Graphics

Torque does not perform its own graphics rasterization; instead, it uses the OpenGL graphics API. Torque includes a utility library that extends OpenGL to support higher-level primitives and resources.

Torque has a collection of utility functions that add support for complex primitives and resources like fonts and bitmaps and that add simple functions for more easily managing textures and 2D rasterization.

Torque includes a texture manager that tracks the loading and unloading of all textures in the game. Only one instance of a texture is ever loaded at a given time; after loading it is handed off to OpenGL. When the game switches graphics modes or video devices, the texture manager can transparently reload and redownload all the game's textures.

Torque supports several bitmap file types: PNG, JPEG, GIF, BMP, and the custom BM8 format, an 8-bit color texture format used to minimize texture memory overhead.

The GUI library manages the user interface of Torque games. It is designed specifically for the needs of game user interface development. The `Canvas` object is the root of the active GUI hierarchy. It dispatches mouse and keyboard events, manages update regions and cursors, and calls the appropriate render methods when it is time to draw the next frame. The `Canvas` keeps track of content controls, which are separate hierarchies of controls that render from bottom to top. The main content control is a screen in the shell that can be covered by any number of floating windows or dialog boxes.

A Profile class maintains common instance data across a set of controls. Information such as font face, colors, bitmaps, and sound data are all stored in instances of the Profile class, so that they don't need to be replicated on each control.

A Control class is the root class for all the GUI controls in the system. A control can contain any number of child controls. Each control maintains a bounding rectangle in the coordinate system of its parent control. The Control class processes input events, rendering, and mouse focus, and coordinates automatic sizing.

3D Rendering

The Torque library has a modular, extensible 3D world rendering system. Game subclasses first define the camera orientation and field of view and then draw the 3D scene using OpenGL drawing commands. A class manages the setting up of the viewport, as well as the model view and projection matrices. A function returns the viewing camera of the current control object (the object in the simulation that the player is currently controlling), and then the engine calls the client scene graph object to render the world.

On the client, a scene graph library is responsible for traversing the world scene and determining which objects in the world should be rendered given the current camera position, while on the server, it determines what objects should be sent to each client based on that client's position in the world. The world is divided into zones, which are volumes of space bounded by solid areas and portals. The outside world is a single zone, and interior objects can have multiple interior zones. The engine finds the zone of a given 3D point and which object owns that zone. The engine then determines which zone or zones contain an object instance. At render time the scene is traversed starting from the zone that contains the camera, clipping each zone's objects to the visible portal set from the zones before it. The engine also performs the scoping of network objects, deciding whether a given object needs to be dealt with by a client.

Every world object in the scene that can be rendered is derived from a single base class. As the world is traversed, visible objects are asked to prepare one or more render images that are then inserted into the current scene. Render images are sorted based on translucency and then rendered. This system permits an interior object with multiple translucent windows to render the building first, followed by other objects, followed by the building's windows. Objects can insert any number of images for rendering.

Terrain

The terrain library deals with objects that render a model of the outside world. It contains a `sky` object that renders the outside skybox, animates and renders cloud layers, and applies the visible distance and fog distance settings for when the world as a whole is rendered. The `sky` object also generates the vertical fog layers and sends them into the `SceneGraph` object for rendering. The TerrainBlock class provides a single 256×256 infinitely repeating block of heightfield terrain. Heightfield data is stored and loaded by the resource manager so that a single terrain data file can be shared between server and client.

The terrain is textured by blending base material textures with program code into new material textures and then mapping those across multiple terrain squares based on the distance from the square. The Blender class performs the blending of terrain textures and includes a special assembly version to speed things up when executing on x86 architectures.

Team LRN

Water is dynamically rendered based on distance, making nearby water more tessellated and detailed. Water coverage of an area can be set to seed fill from a point on the surface, allowing the water to fill a depression to form a lake without leaking outside the corners.

Interiors

The interior library manages the rendering, collision, and disk-file services for interior objects, such as buildings. An interior resource class manages the data associated with a single definition of an interior, and multiple instances may exist at any one time. Interiors manage zones for the scene graph and may have subobjects that render a mirrored view. A light manager class generates lightmaps for all currently loaded interiors. Lightmaps are shared among instances whenever possible. Interior resources are built and lit by an interior importer utility. The source files are Quake-style .map files that are little more than lists of convex physical constructive solid geometry "brushes" that define the solid areas of the interior. Special brushes define zone portal boundaries and objects like lights.

Shapes and Animation

A library manages the display and animation of shape models in the world. This library's shape resource class can be shared between multiple shape instances. The shape class manages all the static data for a shape: mesh data, animation keyframes, material lists, decal information, triggers, and detail levels. An instance class manages animation, rendering, and detail selection for an instance of a shape. The instance class uses the thread class to manage one of the concurrently running animations on an instance. Each thread can be individually advanced in time or can be set on a time scale that is used when all threads are advanced. A thread can also manage transitions between sequences.

Animation sequences can be composed of node/bone animation (for example, joints in an explosion), material animation (a texture animation on an explosion), and mesh animation (a morphing blob; note that most mesh animations can be accomplished with node scale and rotation animations). Animations can also contain visibility tracks so that some meshes in the shape are not visible until an animation is played.

Networking

Torque was designed from the foundation to offer robust client/server network simulation support. The networking design of Torque was driven by the need for superior network performance over the Internet. Torque addresses three fundamental problems of real-time network programming: limited bandwidth, packet loss, and latency. For a more detailed, if somewhat outdated, description of the Torque network architecture, see "The Tribes II Engine Networking Model," an article by Tim Gift and Mark Frohnmayer, at the GarageGames site (http://www.garagegames.com). An instance of a Torque game can be set up as a dedicated server, a client, or both client and server. If the game is both client

Team LRN

and server, it still behaves as a client connected to a server, but the netcode has a short-circuit link to other netcode in the same game instance, and no data goes out to the network.

Bandwidth is a problem because of the large, open terrain environments Torque supports, as well as the large number of clients Torque can handle—up to 128 or more per server, which means that there is a high probability that many different objects can be moving and updating at the same time. Torque uses several strategies to maximize available bandwidth.

- It sends updates to what is most important to a client at a greater frequency than it updates data that is less important.
- It sends only the absolute minimum number of bits needed for a given piece of data.
- It only sends the part of the object state that has changed.
- It caches common strings and data so that they need only be transmitted once.

Packet loss is a problem because the information in lost data packets must somehow be retransmitted, yet in many cases the data in the dropped packet, if sent again directly, will be stale by the time it gets to the client.

Latency is a problem in the simulation because the network delay in data transmission makes the client's view of the world perpetually out of sync with the server. Twitch-style FPS games, for which Torque was initially designed, require instant control response in order to feel anything but sluggish. Also, fast-moving objects can be difficult for highly latent players to hit. In order to solve these problems, Torque employs the following strategies:

- *Interpolation* is used to smoothly move an object from where the client thinks it is to where the server says it is.
- *Extrapolation* is used to guess where the object is going based on its state and rules of movement.
- *Prediction* is used to form an educated guess about where an object is going based on rules of movement and client input.

The network architecture is layered: At the bottom is the OS/platform layer, above that the notify protocol layer, followed by the `NetConnection` object and event management layer.

Using Torque in This Book

As you've seen, the Torque Game Engine is powerful, feature rich, flexible, and controllable. What we will do in this book is create all of the different elements of the game that we'll need and then write game control script code to tie it all together.

All of the program code, artwork, and audio resources you will need are included on the companion CD, along with the tools needed to manipulate them and create your own.

Team LRN

At first glance that may not seem to be too daunting a task. But remember, we will be wearing *all* of the game developer hats. So we will be creating our own models (players, buildings, decorations, and terrains), recording our own sound effects, placing all of these things in a virtual world of our own fabrication, and then devising game rules and their scripted implementations to make it all happen.

Daunted yet?

Hey, it's not going to be *that* hard. We've got Torque!

The CD

There are several different setup options available from the CD. The simplest and most complete is the Full Install. The most minimal installation will install the Torque Engine Executable and the appropriate file paths for a sample game, with supporting scripts.

Installing Torque

If you want to install only the Torque Game Engine, without the various chapter files, extra utilities, or demo games, then do the following:

1. Browse to your CD in the \Torque folder.
2. Locate the Setup.exe file and double-click it to run it.
3. Click the Next button for the Welcome screen.
4. Click the Next button for the Destination screen, taking the default program group location.
5. At the Select Components screen there is a Full Installation drop-down menu. Select this menu by clicking in it, and change it by selecting Custom Installation. Then click the Next button.
6. From the Components list, select Torque and click the Next button.
7. Select the defaults for the remaining screen, clicking Next for each one.

Moving Right Along

There you go. You now have the basic Torque Game Engine plus a sample game installed. Enjoy!

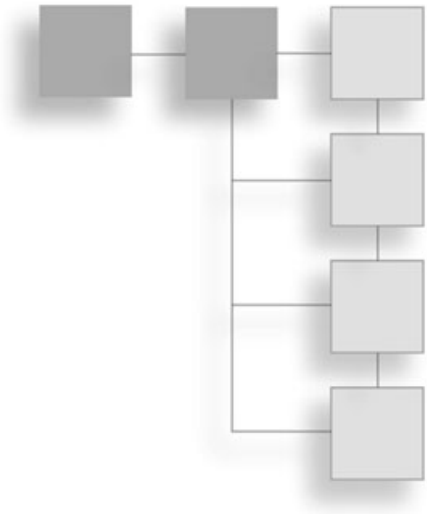
Of course, if you are following along with the game development in this book, you will need to return to the CD and install all the other components when they are needed.

During the chapter, we've looked at computer games from many different angles—the industry, the genres, and the different roles of developers, as well as an exploration into what things make a game engine work and how they relate to each other.

In the next chapter, we'll get into the basics of programming. We'll use the Torque Engine itself to run our example programs as we work through the chapter. This will develop skills we'll need in later chapters when we start delving into real game programming scripts.

Team LRN

CHAPTER 2



Introduction to Programming

My intent with this chapter is to help you understand programming concepts and techniques and leave you with a foundation upon which you can build more advanced skills. By the end of this chapter, you will be proficient with a powerful programming editor; understand how to create, compile, and run programs you've written yourself; have a reasonable sense of programming problem-solving methods; and become familiar with valuable debugging tips and techniques.

UltraEdit-32

To write our programs, we will need to use a *text editor*, or *programming editor*. This kind of editor differs from a word processor, which is what most people use for writing documents, books, memos, and church bulletins.

A good programming editor has several useful features:

- A project feature that allows you to organize your source files
- A fully featured grep (find, search, and replace) capability
- Syntax highlighting
- A function finder or reference
- Macro capability
- Bookmarks
- Text balancing or matching

I use a shareware editor called *UltraEdit-32* (UltraEdit), written by Ian D. Meade, included on the companion CD for this book. It also has several other useful features that I'll demonstrate later in this chapter.

Team LRN

grep? What Kind of Name Is That?

The name *grep* comes from the UNIX world, where strange and wonderful names and incantations for programs abound. grep is derived from the command string "g/re/p" which first appeared in old line editor programs on early UNIX systems. The "g" meant global, the "re" meant regular expression, and the "p" meant print, as in print to the screen. If you entered that command into the editor's command line, you were telling the editor to globally search, using regular expression syntax, and then print the results—and the expression would then follow those characters. Eventually that command string was migrated outside of the editor program and incorporated into a command that was usable from the UNIX command shell as a way of specifying how to look and what to look for when you are searching files that contain a particular piece of text. Over time, the name grep became synonymous with searching files for embedded text and has become a common term in the programming world, even in non-UNIX environments. Now it is often used as a verb meaning "search for text in files."

Program Setup and Configuration

After you insert the companion CD into your computer's CD drive, use Windows Explorer to browse your way on the CD into the folder called 3DGPAi1. Find setup.exe, double-click it, and follow the installation instructions.

Next, browse your way on the CD into the folder called UltraEdit-32. Find setup.exe, double-click it, and follow the installation instructions.

Finally, browse your way on the CD into the folder called UE Tools. Find setup.exe and double-click it to run it and follow the installation instructions. This will install UE Project Maker in the 3DGPAi1 folder on your C drive. This tool will automatically generate project files that you can use with UltraEdit-32.

Setting Up Projects and Files

note

Use the UE sample folder in the 3DGPAi1 folder.

Like any decent editor environment, UltraEdit-32 allows us to organize the files we want to work with using a *projects* concept. You can create, in UltraEdit-32, virtual folders and save links to your files in these folders. By doing this, you can create a quick and convenient access channel to files that are stored anywhere, even somewhere on the network! Setting up your projects can be a bit tedious, however, depending on your needs. To help you with setup, I have written a utility called *UltraEdit Project Maker* (UEPM), which is included on the companion CD. I'll tell you more about using UEPM later, but right now, let's dive in and manually set up a project.

Team LRN

Configuring UltraEdit

To configure UltraEdit, follow these steps:

1. Launch UltraEdit by selecting Start, Program Files, UltraEdit, UltraEdit-32 Text Editor.
2. Close any open files you may have in UltraEdit by selecting Window, Close All Files.
3. In UltraEdit, select View, Views/Lists, File Tree View. A new docked window will appear on the left side (see Figure 2.1). This is the File Tree View.
4. In the File Tree View there is a drop-down combo box (it has a down-pointing arrow at its right end; see Figure 2.2). If the text in the combo box does not say "Project Files," then click the arrow on the right and select Project Files from the list that drops down. When the name has been set to Project Files, we refer to this as the Project View.
5. Right-click in the white area of the Project View to get a pop-up menu. Select Show Names Only.
6. If the Project View is free-floating (not docked), then repeat the right-click and this time select Allow Docking if it isn't already selected. Then click and hold (grab) the colored bar at the top of the File List View/Project View window where it says "File List View" and drag it to the left side of your UltraEdit window such that the colored bar remains in the dark gray space, but the left side of the view window disappears off the left side of the UltraEdit window. You should see the outline of the view window change from a wide gray line to a thin black line. Let go of the mouse button and the view will be docked.

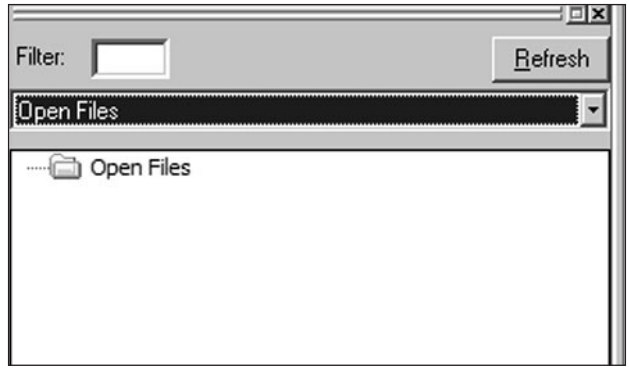


Figure 2.1 Locating the File Tree/Project View.

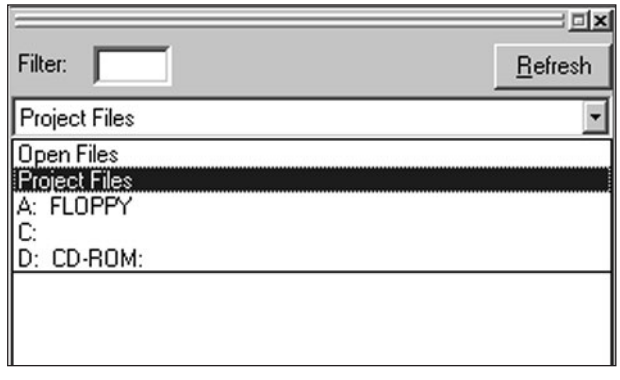


Figure 2.2 Changing the File List View to the Project View.

7. Select the menu item Project, New Project/Workspace. Browse your way to C:\3DGPAi1\UESampleProject. A Save dialog box will appear. Type in the project name (**uesample**), and make sure you have the Project Files type selected in the combo box of the dialog box. Next, the Project dialog box will appear. If you are given an alert that tells you the file already exists, and asks if you want to save, click "Yes".
8. Click the Relative Paths and Relative to Project File check boxes and make sure they are checked.
9. Click New Group and type in **SubFolder** and then click on the OK button. The SubFolder entry will appear in the list.
10. Select the SubFolder entry so that it is highlighted, and then click New Group and type in **SubSubFolder**, then click on the OK button. The SubSubFolder entry will appear in the list nested under SubFolder. You may need to click on the SubFolder icon to make the plus sign appear next to SubFolder, and then click on the plus sign to ensure that SubSubFolder appears nested inside.
11. Select the root entry (it's marked by the [-] symbol). Next click on the New Group button and type in **SubFolderTwo**. The SubFolderTwo entry will appear in the list.
12. Double-check your entries and compare the results with Figure 2.3. Click Close to make the dialog box go away.

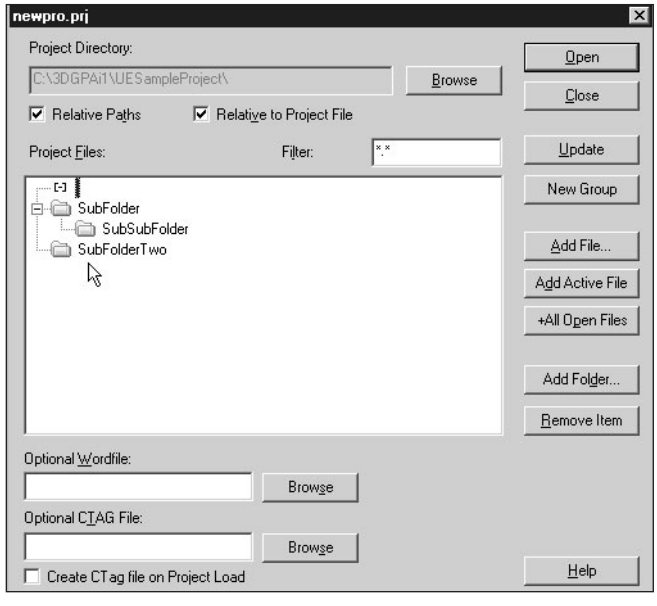


Figure 2.3 Project dialog box with folder hierarchy.

13. Using the menu item File, Open, browse your way to C:\3DGPAi1\UESampleProject and open the file called sample file 1.txt. Do the same for C:\3DGPAi1\UESampleProject\sample file 2.txt. You should now have only these two files open.
14. Open the Project dialog box again, by selecting Project, File/Settings, and click the root entry to select it.
15. Click +All Open Files. The two open files will be added to the project's file list at the root level. Close the Project dialog box.

16. Close both of your open files.
17. Next, open C:\3DGPAi1\UESampleProject\SubFolder\sample file 3.txt and C:\3DGPAi1\UESampleProject\SubFolder\sample file 4.txt.
18. Now reopen the Project dialog box, select the SubFolder entry, and click +All Open Files.
19. Close all of your open files.
20. Repeat steps 18 and 19 for the files located in C:\3DGPAi1\UESampleProject\SubFolderTwo and C:\3DGPAi1\UESampleProject\SubFolder\SubSubFolder, ensuring that you add the file links in the appropriate project folder.

After following these steps, you should have a Project Setup dialog box that looks like Figure 2.4, and your Project View should look like Figure 2.5. You may need to click on the plus sign in front of the folder entries in order to expand the folders to match the view in the figure.

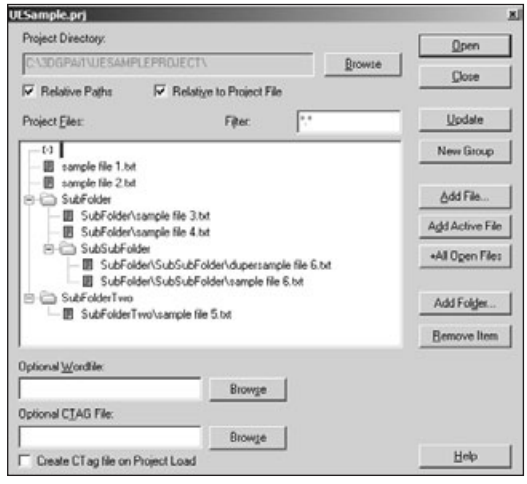


Figure 2.4 Final form of the Project/Workspace Setup dialog box.

As the saying goes, there is more than one way to skin a cat, and in this case there are other ways to set up your project. You can do it all from within the Project/Workspace dialog box using the Add File button. You can also use the Add Active File button to add whatever file is currently the one being edited in UltraEdit. You can experiment and find the method that works best for you. I tend to use a combination of All Files and Add Active File, depending on my needs at the time.

Go ahead and open a few files and close them again, to get a feel for how the Project View works.

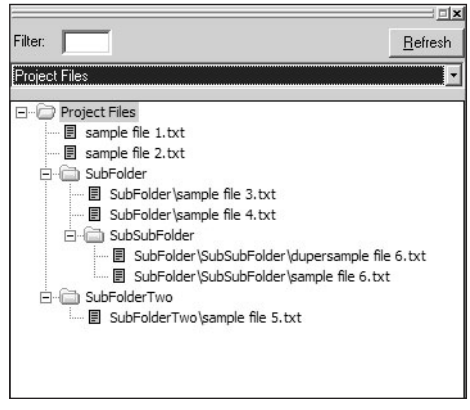


Figure 2.5 Final form of the Example Project View.

Search and Replace

The search capabilities of UltraEdit are quite extensive and thorough. I'm going to focus on the few most important: finding specific text, finding specific text and replacing it, jumping to a line number, and advanced searching using

wildcards and patterns. To practice the various features, open the UESample project, and open the file called sample file 1.txt. It has some text extracted from an early revision of Chapter 1 that we can hack away at.

Find

Select the Search, Find menu item, and you should get the Find dialog box (see Figure 2.6). Make sure the option check boxes match the ones in Figure 2.6. Now, type in the word you want to find, then click the OK button. The Find dialog box will go away, your text insertion point will jump to the first found instance of the word you want, and the word will be highlighted. Try this using the word "indie". See that?

Okay, now get your Find dialog box back and try doing this with the various options. Notice that the Find operates on the currently active file in the editor. Check out the various options, like searching "down" the file and then searching back "up" the file. Change your search word to "INDIE" (all capital letters) and then try your search again. Note that the Find still locates the word. Now try it with the Match Case option checked. Notice that you get an error message: Search String Not Found.

When searching, you will often have more than one match to your search criteria. If you are not using the List Lines option, then you can move through each match in the text by using Search, Find Next to continue to find matching terms as you move toward the end of the file (down). Using Search, Find Prev will do the same thing moving toward the start of the file (up). However, you will probably want to quickly get acquainted with using the keyboard shortcut F3 for Find Next and Ctrl+F3 for Find Prev.

Tip

A quick and convenient way to search for other occurrences of a word that is already written and visible in the active window is to highlight the word (double-click it), press Ctrl+F (the shortcut for Find), and then press Enter. The insertion point will jump to the next occurrence of the word. Then keep pressing F3 to move to the next, and the next, and the next, ad infinitum. UltraEdit will keep starting over from the beginning of the file until it finds no more matches.

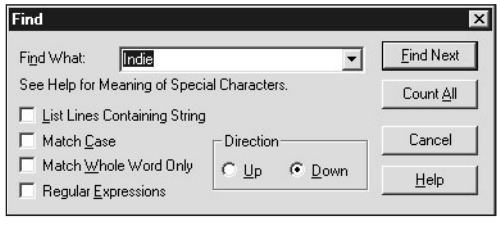


Figure 2.6 The Find dialog box set for a basic search.

A feature of the Find dialog box that I think is particularly useful is the List Lines Containing String option. With this checked, all instances of the word you are looking for will be listed as complete lines in a separate window. Try it by searching for the word "action" with case sensitivity turned off. This should give you a window with a list of lines in it. Each line contains at least one instance

of the search term you used. If you double-click a line, you will see the text and insertion point in your edit window jump to where that line is located and highlight the line.

Special Find Characters

When using Find, there are some things you may want to search for that are not normal alphanumeric characters or punctuation marks—the end of a line, for example.

These are handled by using special characters that are a combination of an *escape* character and a symbol. The escape character is the caret ("^"; you get this when you hold down the Shift key and type the number "6" on North American keyboards) and is paired with a symbol that is a normal character. Whenever Find sees the combination of the caret in front of a character, it knows it is doing a special character search.

Of course, the first special character is the caret itself; otherwise we would never be able to do a search for a caret in text. Look at the following table for a list of the most common special Find characters.

These do *not* require you to turn on the Regular Expressions switch in the Find dialog box, although they are the same as some of the Regular Expressions entries.

Special Characters Used in a Basic Find Function

Special Symbol	What the Program Looks For
^^	caret character ("^"; sometimes called Up Arrow)
^s	highlighted text (only while a macro is running)
^c	contents of the Clipboard (only while a macro is running)
^b	page break
^p	new line (carriage return and line feed) (Windows/DOS files)
^r	new line (carriage return only) (Macintosh files)
^n	new line (line feed only) (UNIX files)
^t	tab character

Replace

Select the Search, Replace menu item, and you should get the Replace dialog box (see Figure 2.7). This dialog box is similar to the Find dialog box, but with more options and a field in which to enter the replacement text.

Team LRN

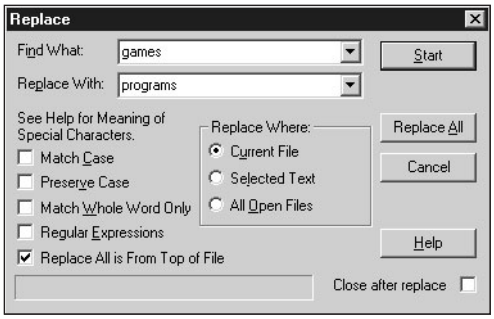


Figure 2.7 The Replace dialog box set for a basic search-and-replace operation.

Find in Files

The Find in Files feature is UltraEdit's closest implementation of grep, which I mentioned earlier in the chapter. The basic Find in Files capability allows you to specify what word or phrase you are looking for and where to look for it in files other than the one you are currently editing (the *active* file). Figure 2.8 shows the Find in Files dialog box. You'll notice that you can specify one of three different sets of files to search.

First, you can search through the *Listed* files. This means you can specify a file name search pattern with extension and a folder to look in. This is quite similar to the built-in Windows Search or Find feature. You can use wildcards to fine-tune which files will be checked. Searching with the In Files/Types box set to "new*.txt", for example, will search inside files with the names newfile.txt, new_data.txt, and so on. Setting the pattern to "*.*" will cause the program to search inside every file it finds in the specified folder. If you have the Search Sub Directories box checked, then it will also look inside every file inside every folder contained in the specified folder.

When the program finds a match in the file with the word you are looking for, it will print a listing at the bottom of the UltraEdit window containing a reference to the file where the word was found, plus the line in which it was found. If you double-click the line in the bottom window, UltraEdit will open the file and position the line in your window for viewing.

Next, you can search only in the *Open Files*—that is, only within the files that are currently open in the editor. If you click the Open Files radio button in the Search In: box, you see that now you only enter the word to search for; you don't need to specify file names or a folder.

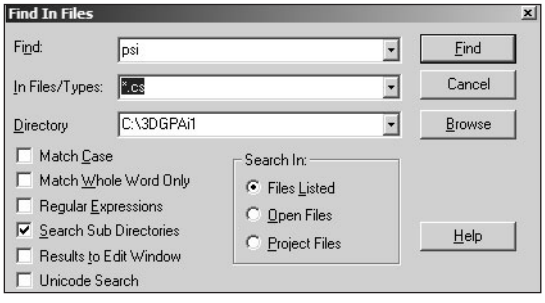


Figure 2.8 The Find in Files dialog box.

Finally, the method I use the most is to search in *Project Files*. With this option checked, the program will search through all of the files in the project you currently have open—and only those files. It doesn't matter whether the files are open or not.

grep

The grep capability in UltraEdit (also see the sidebar earlier in this chapter) is an advanced way of finding text within files and replacing it with other text when desired. You can use it in Search-related topics covered so far by putting a check mark in the Regular Expressions box—then Find will operate using standard UNIX-like grep or the older UltraEdit-specific form of grep.

You can configure UltraEdit to use its own grep syntax or the UNIX-style syntax in the configuration menu. Select the Advanced, Configuration menu item and then select the Find tab. Change the check box labeled UNIX-style Regular Expressions to suit your taste.

UltraEdit-Style grep Syntax

Table 2.1 shows the available UltraEdit-style grep functions. Let's do a few example grep searches to get a feel for how it works. Use the file sample file 1.txt from the UESample project to do the searches. For this section make sure you have the UltraEdit configuration setting for UNIX style Regular Expressions turned off.

Let us suppose that we want to find some reference to dungeons in games in the sample file. We'll grep for (notice that I'm verbing the noun here!) the term "game*dungeon".

Press Ctrl+F to bring up the Find dialog box, and then make sure the Regular Expressions box is checked. Type in the search term **game*dungeon**, and click the Find Next button. The string it finds starts with "game" and ends with "dungeon". The words that appear in between were inconsequential to the search, because the asterisk means that the search program will match any string of characters of any length between the words "game" and "dungeon", as long as it doesn't encounter a new line character or a carriage return. Try it again, but this time use the term "computer*game" and see what you find. Remember that you can use F3 as a shortcut to find the next match.

The operator that is the same as the asterisk, only different, is the question mark ("?"). Instead of matching any number of any characters, it will match only one instance of any character. For example, "s?n" matches "sun", "son", and "sin" but not "sign" or "soon".

Here are some more examples of how the matching criteria work:

Be+st	will find "best", "beest", "beeeest", and so on, but will not find "bst".
[aeiou]	will find every lowercase vowel.
[,.?]	will find a literal ",", ".", or "?".
[0-9a-z]	will find any digit or lowercase letter.
[~0-9]	will find any character except a numeral (the tilde ["~"] means to *not* include whatever follows).

Table 2.1 UltraEdit-Style grep Syntax

Symbol	Purpose
%	Matches the start of line. Indicates the search string must be at the beginning of a line but does not include any line terminator characters in the resulting string selected.
$	Matches the end of line. Indicates the search string must be at the end of a line but does not include any line terminator characters in the resulting string selected.
?	Matches any single character except newline.
*	Matches any number of occurrences of any character except newline.
+	Matches one or more instances of the preceding character. At least one occurrence of the character must be found. Does not match repeated newlines.
++	Matches the preceding character/expression zero or more times. Does not match repeated newlines.
^b	Matches a page break.
^p	Matches a newline (CR/LF) (Windows/DOS Files).
^r	Matches a newline (CR Only) (Mac Files).
^n	Matches a newline (LF Only) (UNIX Files).
^t	Matches a tab character.
[ ]	Matches any single character, or range in the brackets.
^{A^}^{B^}	Matches expression A OR B.
^	Overrides the following regular expression character.
^(…^)	Brackets or tags an expression to use in the replace command. A regular expression may have up to nine tagged expressions, numbered according to their order in the regular expression. The corresponding replacement expression is ^x, for x in the range 1-9. Example: If ^(h*o^) ^(f*s^) matches "hello folks", ^2 ^1 would replace it with "folks hello".

UNIX-Style Syntax

The UNIX-style syntax is used in the same way as the UltraEdit-style, but is different in many ways. The advantages of using the UNIX style are:

- It is somewhat of a standard, so you may be familiar with it from elsewhere.
- It has more capabilities than the UltraEdit syntax.
- At some point in the future it may be the only syntax for grep supported by UltraEdit, when the program's author decides to stop supporting the old UltraEdit-style.

You can see the differences by checking out Table 2.2. The first obvious difference is that the escape character has changed from the caret to the back slash. Our example searches would be a little different; the asterisk doesn't match any character anymore, now it matches any number of occurrences of the character that appears just before it. Also, now we use the period "." to match any single character instead of the question mark.

Team LRN

Before proceeding, make sure you have your editor set to use the proper UNIX-style syntax in the Advanced, Configuration menu under the Find tab.

Now—to go back to our dungeon games example, the way the search term in UNIX-style grep syntax would look is "game.*dungeon".

Compare these examples with the ones for the UltraEdit-style:

be+st	matches "best", "beest", "beeeest", and so on, BUT NOT "bst".
be*st	matches "best", "beest", "beeeest", and so on, AND "bst".
[aeiou]	matches every lowercase vowel.
[,.?]	matches a literal ",", ".", or "?".
[0-9a-z]	matches any digit, or lowercase letter.
[^0-9]	matches any character except a digit (^ means *NOT* the following).

Table 2.2 UNIX-Style grep Syntax

Symbol	Purpose
\	Indicates the next character has a special meaning. "n" on its own matches the character "n". "\n" matches a linefeed or newline character. See examples below (\d, \f, \n).
^	Matches or anchors the beginning of line.
$	Matches or anchors the end of line.
*	Matches the preceding character zero or more times.
+	Matches the preceding character one or more times. Does not match repeated newlines. Matches any single character except a newline character. Does not match repeated newlines.
(expression)	Tags an expression to use in the replace command. A regular expression may have up to 9 tagged expressions, numbered according to their order in the regular expression. The corresponding replacement expression is \x, for x in the range 1-9. Example: If (h.*o) (f.*s) matches "hello folks", \2 \1 would replace it with "folks hello".
[xyz]	A character set. Matches any characters between brackets.
[^xyz]	A negative character set. Matches any characters NOT between brackets.
\d	Matches a number character. Same as [0-9].
\D	Matches a non-number character. Same as [^0-9].
\f	Matches a form-feed character.
\n	Matches a linefeed character.
\r	Matches a carriage return character.
\s	Matches any white space including space, tab, form-feed, and so on, but not newline.
\S	Matches any non-white space character but not newline.
\t	Matches a tab character.
\v	Matches a vertical tab character.
\w	Matches any word character including underscore.
\W	Matches any non-word character.
\p	Matches CR/LF (same as \r\n) to match a DOS line terminator.

Bookmarks

One feature I use quite frequently is the Bookmark capability. Its purpose is to help you find your way around large files quickly. When you are working in an area that you think you may need to come back to later, just set a bookmark, and then when you are working in another place in your document, you can use the Goto Bookmark command to jump through each bookmark you've set until you find the one you want. This sure beats scrolling through all your open files looking for that one spot you worked on two hours ago!

To set a bookmark, click your mouse on a line of text and then select the menu item Search, Toggle Bookmark. The line where the bookmark is set will be highlighted in a different color (See Figure 2.9). In the figure, the lower highlighted line is the bookmarked line.

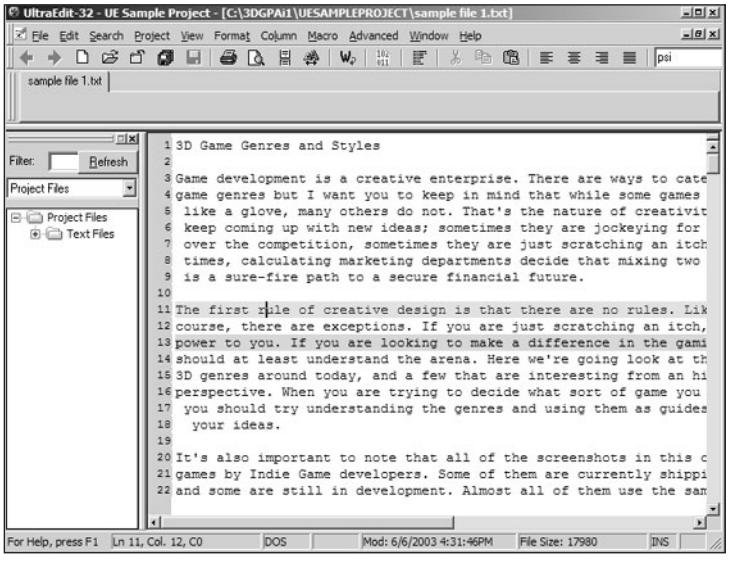

UltraEdit-32 - UE Sample Project - [C:\3DGPAi1\UESAMPLEPROJECT\sample file 1.txt]
File Edit Search Project View Format Column Macro Advanced Window Help
sample file 1.txt
Filter: Refresh
Project Files
Project Files
Text Files
1 3D Game Genres and Styles
2
3 Game development is a creative enterprise. There are ways to cate
4 game genres but I want you to keep in mind that while some games
5 like a glove, many others do not. That's the nature of creativit
6 keep coming up with new ideas; sometimes they are jockeying for
7 over the competition, sometimes they are just scratching an itch
8 times, calculating marketing departments decide that mixing two
9 is a sure-fire path to a secure financial future.
10
11 The first rule of creative design is that there are no rules. Lik
12 course, there are exceptions. If you are just scratching an itch,
13 power to you. If you are looking to make a difference in the gami
14 should at least understand the arena. Here we're going look at th
15 3D genres around today, and a few that are interesting from an hi
16 perspective. When you are trying to decide what sort of game you
17 you should try understanding the genres and using them as guides
18 your ideas.
19
20 It's also important to note that all of the screenshots in this c
21 games by Indie Game developers. Some of them are currently shippi
22 and some are still in development. Almost all of them use the sam
For Help, press F1
Ln 11, Col. 12, C0
DOS
Mod: 6/6/2003 4:31:46PM
File Size: 17980
INS

Figure 2.9 Bookmarked text shown in lighter gray.

To remove a bookmark, click your mouse in the highlighted bookmark line, and select Search, Toggle Bookmark again. This will turn off the bookmark for that line.

To remove all bookmarks, select Search, Clear All Bookmarks, and all bookmarks that you previously set will vanish.

tip

If you are using the Project View, when you close your documents, all the bookmarks you've set will be saved, and restored the next time you open that document. This does not happen with documents that are not associated with the Project View.

To navigate between the bookmarks, select Search, Goto Bookmark and your insertion point will jump to the next bookmark in sequence. You can also select Search, Goto Previous Bookmark, to jump in the reverse direction from bookmark to bookmark.

tip

Most commands available in the menus have keyboard shortcuts available. Rather than listing them here, I'll just point you to the menu items. The keyboard shortcuts for the command, if available, are written next to the menu selection. Some menu items, like Clear All Bookmarks, have no shortcut assigned, but don't despair. You can assign your keyboard shortcuts by using the Key Mapping tab in the Advanced, Configuration menu, and following the instructions. Note that the command names in the list are written with their main menu entry as the first part of the command. The Clear All Bookmarks command is written as SearchClearBookmarks. The commands are listed in alphabetical order.

Macros

Macro commands are like shortcuts. You can string together a whole series of tedious editing operations into a group, called a *macro*, that you can invoke at any time later by a simple keystroke, or menu item, or toolbar button.

UltraEdit has two forms of macros—the standard and the Quick Record macro. Let's take a look at both, starting with the Quick Record macro.

Quick Record Macro

The *Quick Record macro* is a bare-bones macro function.

1. Select the Macro, Quick Record menu item (or press Shift+Ctrl+R).
2. Start performing all the editing actions you want recorded. In this case just type in the text **blah blah blah** somewhere.
3. Select Macro, Stop Quick Recording (or press Shift+Ctrl+R again).

Now replay your edit actions over again at any place in your text by simply placing your text insertion point where appropriate, and typing Ctrl+M, or selecting the Macro, Play Again menu item.

You can only ever have one Quick Record macro—each time you record one, it replaces the earlier recording.

Standard Macro

Standard macros are a bit more complex. The procedure for recording them is somewhat similar, but you can assign them to key combinations of your choice, or to menus, or even to toolbar buttons. This gives you much more flexibility than the Quick Record macro, but at the cost of a bit of setup twiddling, of course.

Let's make a couple of standard macros. One will insert the words **This is cool** and the other will jump to the beginning of whatever line the insertion point is on, then capitalize the first word, put a period at the end, and then insert the phrase **Capital Idea!** after the period.

Team LRN

1. Place your insertion point in a blank line somewhere.
2. Select the Macro, Record menu item.
3. In the Macro Name box, give it a name, something like "InsertCool".
4. Click the mouse in the HotKey edit box to the right of where it says "Press New Key" and then press and hold Shift+Ctrl+I.
5. Click the OK button.
6. Type in the phrase **This is cool**.
7. Select Macro, Stop Recording.
8. Place your insertion point at the end of the line with the phrase "This is cool" in it.
9. Select the Macro, Record menu item.
10. In the Macro Name box, give it a name, something like "MakeCapital".
11. Click the mouse in the HotKey edit box to the right of where it says "Press New Key", and then press and hold Shift+Ctrl+M.
12. Click the OK button.
13. Type the following key sequence, one at a time (don't type the text in parentheses):

    Home

    Shift+Ctrl+Right Arrow

    F5

    End

    . (that's a period)

    spacebar
14. Now type the phrase **Capital Idea!**
15. Now select the Macro, Stop Recording menu item.

There, that's done. So now let's test it out.

First, find or create a blank line, place your insertion point on it, and then press Shift+Ctrl+I. See the text that gets inserted? Okay, now leave your text insertion point in that new text, anywhere, and then press Shift+Ctrl+M. You should end with a line that says, "This is cool. Capital Idea!", with the same capitalization. Macros *are* cool!

UltraEdit Review

So now you've seen how to use what are, in my opinion, the most important editing features of UltraEdit—grep (Find and Replace), macros, and bookmarks—and you've seen how UltraEdit can be configured in a project format to make it easy to use files in an organized fashion.

Team LRN

UltraEdit has a good Help feature that covers all aspects of the program, so I encourage you to use it.

Remember that UltraEdit is an *editor*, not a *word processor*, so there aren't a great deal of formatting features in the program, which is just as well because we are using it to write code and not to write documents or books. The focus is on the steak, not the sizzle.

Speaking of steak, it is now time to get to the meat of this chapter, coming up next!

Controlling Computers with Programs

When you create a computer program, you are creating a set of instructions that tell the computer exactly and completely what to do. Now before you jump all over me and hammer me with comments like, "Well, duh! Of course programming a computer is like telling it what to do," I want you to read the first sentence again. It is not an analogy, and it is not some kind of vague and airy all-encompassing cop-out.

Everything that a computer does, at any time, is decided by at least one programmer. In the vast majority of cases, the computer's instructions—contained in *programs*—are the work-product of hundreds, if not thousands, of programmers. All of the programs that a computer uses are organized and classified in many different ways. The organization helps us humans keep track of what they do, why we need them, how to link one program with another, and other useful things. The computer's operating system is a huge collection of programs designed to work in conjunction with other programs, or sometimes to work alone, but in the context created by other programs.

We leverage the efforts of other programmers when we sit down to program a computer for any purpose. One of the results of many that have gone before is the creation of *programming languages*. Computers operate using a language that is usually unique to each brand and model, called *machine code*. Machine code is designed to directly control the computer's electronics—the hardware. Machine code is *not* very friendly to humans.

To give you an idea, we'll look at an example of machine code that tells a computer using an Intel 80386 chip to add together two numbers and save the result somewhere. What we will do is add A and B together and leave the result in C. To start, A will equal 4 and B will equal 6.

So our formula will be a simple math problem:

```
A=4
B=6
C = A + B
```

The computer machine code looks like this:

```
11000111000001010000000000000000000000000000000000000010000000000000000000000000110001110
00001010000000000000000000000000000000000000011000000000000000000000000010100001000000000
00000000000000000000000000001100000101000000000000000000000000000000010100011000000000000
00000000000000000000
```

Now go ahead and look carefully at that and tell yourself honestly whether you could work with a computer using machine code for longer than, oh, about 12 minutes! My personal best is somewhere around 30 seconds, but that's just me. The number system used here is the *binary* system.

Each one of those 1s and 0s is called a *bit* and has a precise meaning to the computer. This is all the computer actually understands—the ones, the zeros, their location and organization, and when and how they are to be used. To make it easier for humans to read machine code at those rare times when it is actually necessary, we normally organize the machine code with a different number system, called *hexadecimal* (or *hex*), which is a base-16 number system (rather than base-10 like the *decimal* system we use in everyday work). Every 4 bits becomes a hex numeral, using the symbols from 0 to 9 and the letters A to F. We pair two hex numerals to carry the information contained in 8 bits from the machine code. This compresses the information into an easier-to-read and more manageable size. Here is the same calculation written in the hex form of machine code:

```
C7 05 00 00 00 00 04 00 00 00 C7 05 00 00 00 00 06 00 00 00 A1 00 00 00 00 03 05 00 00
00 00 A3 00 00 00 00
```

Much better and easier on the eyes! There are many people who work close to the computer hardware who work in hex quite often, but it still is pretty obscure. Fortunately, there is a human-readable form of the machine code for every microprocessor or computer, which in general is known as *assembly language*. In this case we use words and symbols to represent meaningful things to us as programmers. Tools called *assemblers* convert assembly language programs to the machine code we looked at earlier. Here is the Intel 80386 Assembler version of our little math problem:

```
mov     DWORD PTR a, 4      ; (1)
mov     DWORD PTR b, 6      ; (2)
mov     eax, DWORD PTR a    ; (3)
add     eax, DWORD PTR b    ; (4)
mov     DWORD PTR c, eax    ; (5)
```

Now we are getting somewhere! Let's take a closer look. Lines 1 and 2 save the numbers 4 and 6 in memory somewhere, referenced by the symbols a and b. The third line gets the value for a (4) and stores it in some scratch memory. Line 4 gets the value for b (6), adds it to the 4 in scratch memory, and leaves the result in the same place. The last line moves the result into a place represented by the symbol c. The semicolon tells the assembler tool to ignore what comes after it; we use the area after the semicolon to write commentary

and notes about the program. In this case I've used the comment space to mark the line numbers for reference.

Now that, my friends, is a program! Small and simple, yes, but it is clear and explicit and in complete control of the computer.

As useful as assembly language code is, you can see that it is still somewhat awkward. It is important to note that some large and complex programs have been written in assembly language, but it is not done often these days. Assembly language is as close to the computer hardware as one would ever willingly want to approach. You are better served by using a *high-level language.* The next version of our calculation is in a powerful high-level language called C. No, really! That's the name of the language. Here is our calculation written in C:

```
a=4;      // (1)
b=6;      // (2)c=a+b;    // (3)
```

Now, if you're thinking what I think you're thinking, then you're thinking, "Hey! That code looks an awful lot like the original formula!" And you know what? I think you are right. And that's part of the point behind this rather long-winded introduction: When we program, we want to use a programming language that best represents the elements of the problem we want to solve. Another point is that quite a few things are done for the programmer behind the scenes—there is a great deal of complexity. Also, you should realize that there are even more layers of complexity "below" the machine code, and that is the electronics. We're not even going to go there. The complexity exists simply because it is the nature of the computer software beast. But be aware that the same hidden complexity can sometimes lead to problems that will need to be resolved. But it's not magic—it's software.

The C language you've just seen is what is known as a *procedural* language. It is designed to allow programmers to solve problems by describing the procedure to use, and defining the elements that are used during the procedure. Over time, programmers started looking for more powerful methods of describing problems, and one such method that surfaced was called *object-oriented programming* (OOP).

The simplest point behind OOP is that programmers have a means to describe the relationships between collections of code and variables that are known as *objects.* The C language eventually spawned a very popular variant called C++. C++ includes the ability to use the original C procedural programming techniques, as well as the new object-oriented methods. So we commonly refer to C/C++, acknowledging the existence of both procedural and object-oriented capabilities. From here on, in the book, I will refer to C/C++ as the general name of the language, unless I need to specifically refer to one or the other for some detailed reason.

Team LRN

Programming Concepts

For the rest of this chapter, we are going to explore basic programming techniques. We will be using Torque Script for all of our code examples and running our little programs in the Torque Engine to see what they do.

Now, we just covered the simple math problem in the previous section. I showed you what the program looked liked in binary machine language, hex machine language, assembly language, and finally C/C++. Well, here is one more version—Torque Script:

```
%a=4;        // (1)
%b=6;        // (2)
%c=%a+%b;    // (3)
```

Notice the similarity to C/C++? Even the comments are done the same way!

As demonstrated, Torque Script is much like C/C++. There are a few exceptions, the most notable being that Torque Script is *typeless* and does not require *forward declarations* of variables. Also, as you can see for yourself in the preceding code, Torque Script requires *scope prefixes* (the percent signs) on its variable names.

Typeless? Forward Declarations? Huh?

In many languages, variables have a characteristic called *type*. In its simplest form, a type merely specifies how much memory is used to store the variable. Torque Script doesn't require you to specify what type your variable has. In fact, there is no way to do it!

Forward declarations are a construct whereby the programmer must first indicate, usually at the beginning of a file or a subroutine block, what variables will be used and what their types are. Torque Script also doesn't require this and again provides no mechanism for using forward declarations.

So now that you know what types and forward declarations are, you can forget about them!

The goal for you to achieve by the end of this chapter is the ability to put together simple programs to solve problems and have enough understanding of program techniques to make sensible decisions about the approaches to take.

How to Create and Run the Example Programs

There is an ancient and well-understood programming cycle called the *Edit-Compile-Link-Run* cycle. The same cycle applies with Torque, with the exception being that there is no link step. So for us, it can be thought of as the *Edit-Compile-Run* cycle. A further wrinkle to toss in is the fact that Torque will automatically compile a source file (that is, a program

file that ends with .cs) into the binary byte code file (ends with .cs.dso), if there is no binary version of the file, or if the source file has changed since the last binary was created.

So I guess my point is, for us the cycle can now be regarded as the *Edit-Run* cycle.

- Put all user programs in the folder C:\3DGPAi1\CH2 as filename.cs where "filename" is a name you've either made up yourself or one that I've suggested here in the book.
- Run from command shell tge -CH2 filename.cs.

Hello World

Our first program is somewhat of a tradition. Called the *Hello World* program, it is used as an early confidence builder and test program to make sure that the Gentle Reader (that would be you, if you are reading this book!) has everything in place on his computer to successfully edit, compile, and run a program.

So, assuming that you've installed both UltraEdit and the Torque Game Engine, use your newly learned UltraEdit skills to create a new file with the name HelloWorld.cs and save it in the folder C:\3DGPAi1\CH2. Type into the file these lines of code:

```
// ========================================================================
//  HelloWorld.cs
//
//  This module is a program that prints a simple greeting on the screen.
//
// ========================================================================

function main()
// ------------------------------------------------
//     Entry point for the program.
// ------------------------------------------------
{
   print("Hello World");
}
```

Save your work. Now, use the following procedure to run your program:

1. From your Windows Start menu select the Start, Run.
2. Type **command** in the edit box offered. This will open a Command window or MS-DOS Prompt window.
3. In the new window at the command prompt (the blinking cursor), type **cd C:\3DGPAi1** and then press Enter.

4. Next, type **tge -ch2 HelloWorld.cs**. A Torque window will open up, and you should see something like Figure 2.10, with "Hello World" in yellow at the upper left of the screen. Cool, huh?

tip

If you don't get the expected result on your screen, then look in the log file, named console.log, located in the C:\3DGPAi1 folder. If there were any errors in your program, diagnostic information will be deposited there. It might be something as simple as a typo in the file name.

Let's have a closer look at the code. The first thing you will notice is this stuff:

```
// ============================================================================
//  HelloWorld.cs
//
//  This module is a program that prints a simple greeting on the screen.
//
// ============================================================================
```

This is the *module header block*. It is not executable code—it's what we call a *comment*. The double-slash operator ("//") tells the Torque Engine to ignore everything from the slashes to the end of the line.

So if the engine ignores it, why do we use it? Well, it's included in order to document what the module does so that later when we've completely forgotten the details, we can easily refresh our memory. It also is included to help other programmers who may come along and need to understand the module so they can add new features or fix bugs.

Figure 2.10 Output of the Hello World program.

There are no real rules regarding the format of these headers, but most programmers or development shops have some sort of template that they want followed. At a minimum, the header should include the module file name, copyright notices, and a general description of what the code in the module is for. Sometimes we might include other details that are necessary for a person to understand how the module is used.

Then there is this part:

```
function main()
```

That is *executable code*. It is the declaration of the function block called `main()`. Following that, there is this:

```
// -------------------------------------------------
//     Entry point for the program.
// -------------------------------------------------
```

This is the *function header*. The function header is included in order to describe the specifics of a function—what it does, how it does it, and so on. In this case it is fairly simple, but function headers can get to be quite descriptive, as you'll see later. Again, this is not executable code (note the double slash) and is not required to make your program work. The dashes could just as well be stars, equals signs, or nothing at all. It is good practice to always use function headers to describe your functions.

Finally comes this:

```
{
   print("Hello World");
}
```

That would be the *function body*—the guts of the function where the work is done. The function body is also sometimes called a *function block*, and more generically (when used in other contexts that you'll see later) called a *code block*.

All sample programs in this chapter must have something called *function main()*. Don't worry about why that is for the moment—we'll get into that later. Just take it as a given that it is necessary for your programs to work properly.

It is important to note the way a *function block* is made. It always begins with the keyword function followed by one or more spaces and whatever name you want it to have. After the name comes the argument list (or parameter list). In this case there are no parameters. Then comes the opening, or left, brace (or curly bracket). After the opening brace comes the body of the function, followed by the closing, or right, brace.

All functions have this same structure. Some functions can be several pages long, so the structure may not be immediately obvious, but it's there.

The actual code that does anything interesting is a single line. As you know by now, the line simply prints the text "Hello World" in the Torque window.

Team LRN

note

Experienced C programmers will recognize the `main` function as the required initial entry point for a C program, although the syntax is slightly different. Torque doesn't require this organization for its scripts—it is purely for learning purposes.

Expressions

When we write program code, most of the lines, or *statements*, that we create can be evaluated. A statement can be a single Torque Script line of any kind terminated by a semicolon, or it can be a *compound statement*, which is a sequence of statements enclosed in left and right braces that acts as a single statement. A semicolon does not follow the closing right brace. Here is an example of a statement:

```
print("Hi there!");
```

Here is another example:

```
if (%tooBig == true) print("It's TOO BIG!");
```

And here is one final example of a valid statement:

```
{
     print("Nah! It's only a little motorcycle.");
}
```

Statements that can be evaluated are called *expressions*. An expression can be a complete line of code or a fragment of a line, but the important fact is that it has a value. In Torque the value may be either a number or text (a string)—the difference is in how the value is used. Variables are explained in the next section, but I'll sneak a few in here without detailed coverage in order to illustrate expressions.

Here is an expression:

```
5 + 1
```

This expression *evaluates* to 6, the value you get when 5 and 1 are added.

Here is another expression:

```
%a = 67;
```

This is an *assignment statement*, but more importantly right now, it is an expression that evaluates to 67.

Another:

```
%isOpen = true;
```

This expression evaluates to 1. Why? Because `true` evaluates to the value 1 in Torque. Okay, so I hadn't told you that yet—sorry about that. Also, `false` evaluates to 0. We can say the statements evaluate to `true` or `false`, instead of 1 and 0. It really depends on whatever makes sense in the usage context. You'll notice that the evaluation of the statement is determined by whatever expression is to the right of the equal sign. This is a pretty hard-and-fast rule.

Consider this code fragment:

```
%a = 5;
if (%a > 1 )
```

What do you figure that the `(%a > 1 )` evaluates to, if `%a` has been set to `5`? That's right—it evaluates to `true`. We would read the line as "if %a is greater than 1." If it was written as `(%a > 10 )`, it would have been `false`, because 5 is not greater than 10.

Another way we could write the second line is like this:

```
if ( (%a > 1 ) == true )
```

It would be read as "if the statement that %a is greater than 1 is `true`." However, the Department of Redundancy Department could have written that example. The first way I showed you is more appropriate.

Just for your information, in the preceding examples, `%a` and `%isOpen` are *variables*, and that's what is coming up next.

Variables

Variables are chunks of memory where values are stored. A program that reads a series of numbers and totals them up will use a variable to represent each number when it's entered and another variable to represent the total. We assign names to these chunks of memory so that we can save and retrieve the data stored there. This is just like high school algebra, where we were taught to write something like "Let v stand for the velocity of the marble" and so on. In that case v is the identifier (or name) of the variable. Torque Script identifier rules state that an identifier must do the following:

- It must not be a Torque Script keyword.
- It must start with an alphabetical character.
- It must consist only of alphanumeric characters or an underscore symbol (_) .

A *keyword* is an otherwise valid identifier that has special significance to Torque. Table 2.3 gives a keyword list. For the purposes of Torque identifiers, the underscore symbol (_) is considered to be an alphanumeric character. The following are valid variable identifiers:

isOpen Today X the_result item_234 NOW

These are not legal identifiers:

5input miles-per-hour function `true` +level

Table 2.3 Torque Script Keywords

Keyword	Description
break	Breaks execution out of a loop.
case	Indicates a choice in a switch block.
continue	Causes execution to continue at top of loop.
default	Indicates the choice to make in a switch block when no cases match.
do	Indicates start of a `do-while` type loop block.
else	Indicates alternative execution path in an if statement.
false	Evaluates to 0, the opposite of true.
for	Indicates start of a `for` loop.
function	Indicates that the following code block is a callable function.
if	Indicates start of a conditional (comparison) statement.
new	Creates a new object data block.
return	Indicates return from a function.
switch	Indicates start of a switch selection block.
true	Evaluates to 1, the opposite of false.
while	Indicates start of a `while` loop.

It's up to you as the programmer to choose the identifiers you want to use. You should choose them to be significant to your program and what it is doing. You should always try to use meaningful identifiers. You should note that Torque is not case-sensitive. Lowercase letters are *not* treated as distinct from uppercase letters.

You assign values to variables with an assignment statement:

```
$bananaCost = 1.15;
$appleCost = 0.55;
$numApples = 3;
$numBananas = 1;
```

Notice that each variable has a dollar sign ("$") preceding it. This is the *scope* prefix. This means that the variable has *global* scope—it can be accessed from anywhere in your program, inside any function, or even outside functions and in different program files.

There is another scope prefix—the percent sign ("%"). The scope of variables with this prefix is *local*. This means that the values represented by these variables are valid only

within a function, and only within the specific functions where they are used. We will delve into scoping in more detail later.

Using our fruit example, we can calculate the number of fruit as follows:

```
[$numFruit = $numBananas + $numApples;
```

And we can calculate the total cost of all the fruit like this:

```
$numPrice = ($numBananas * $bananaCost) + ($numApples * $appleCost);
```

Here is a complete small program you can use to try it out yourself.

```
// ========================================================================
//  Fruit.cs
//
//  This module is a program that prints a simple greeting on the screen.
//  This program adds up the costs and quantities of selected fruit types
//  and outputs the results to the display
// ========================================================================

function main()
// ------------------------------------------------
//     Entry point for the program.
// ------------------------------------------------
{
 $bananaCost=1.15;// initialize the value of our variables
 $appleCost=0.55; //   (we don't need to repeat the above
 $numApples=3;    //   comment for each initialization, just
 $numBananas=1;   //   group the init statements together.)

 $numFruit=0;     // always a good idea to initialize *all* variables!
 $total=0;        // (even if we know we are going to change them later)

 print("Cost of Bananas(ea.):$"@$bananaCost);
             // the value of $bananaCost gets concatenated to the end
             // of the "Cost of Bananas:" string. Then the
             // full string gets printed. same goes for the next 3 lines
 print("Cost of Apples(ea.):$"@$appleCost);
 print("Number of Bananas:"@$numBananas);
 print("Number of Apples:"@$numApples);

 $numFruit=$numBananas+$numApples; // add up the total number of fruits
 $total = ($numBananas * $bananaCost) +
               ($numApples * $appleCost);  // calculate the total cost
```

```
            //(notice that statements can extend beyond a single line)

 print("Total amount of Fruit:"@$numFruit); // output the results
 print("Total Price of Fruit:$"@$total@"0");// add a zero to the end
                                 // to make it look better on the screen
}
```

Save the program in the same way you did the Hello World program. Use a name like fruit.cs and run it to see the results. Note that the asterisk ("*") is used as the multiplication symbol and the plus sign ("+") is used for addition. These *operators*—as well as the parentheses used for evaluation precedence—are discussed later in this chapter.

Arrays

When your Fruit program runs, a variable is accessed in expressions using the identifier associated with that variable. At times you will need to use long lists of values; there is a special kind of variable called an *array* that you can use for lists of related values. The idea is to just use a single identifier for the whole list, with a special mechanism to identify which specific value—or *element*—of the list you want to access. Each value has numerical position within the array, and we call the number used to specify the position the *index* of the array element in question.

Let us say you have a list of values and you want to get a total, like in the previous example. If you are only using a few values (no more than two or three), then a different identifier could be used for each variable, as we did in the Fruit program.

However, if you have a large list—more than two or three values—your code will start to get awkwardly large and hard to maintain. What we can do is use a loop, and *iterate* through the list of values, using the indices. We'll get into loops in detail later in this chapter. Here is a new version of the Fruit program that deals with more types of fruit. There are some significant changes in how we perform what is essentially the same operation. At first glance, it may seem to be more unwieldy than the Fruit program, but look again, especially in the computation section.

```
// ========================================================================
//  FruitLoopy.cs
//
//  This module is a program that prints a simple greeting on the screen.
//  This program adds up the costs and quantities of selected fruit types
//  and outputs the results to the display. This module is a variation
//  of the Fruit.cs module
// ========================================================================

function main()
```

```
// ------------------------------------------------
//     Entry point for the program.
// ------------------------------------------------
{
   //
   // ----------------- Initialization --------------------
   //

   %numFruitTypes = 5; // so we know how many types are in our arrays

   %bananaIdx=0;    // initialize the values of our index variables
   %appleIdx=1;
   %orangeIdx=2;
   %mangoIdx=3;
   %pearIdx=4;

   %names[%bananaIdx] = "bananas"; // initialize the fruit name values
   %names[%appleIdx] = "apples";
   %names[%orangeIdx] = "oranges";
   %names[%mangoIdx] = "mangos";
   %names[%pearIdx] = "pears";

   %cost[%bananaIdx] = 1.15; // initialize the price values
   %cost[%appleIdx] = 0.55;
   %cost[%orangeIdx] = 0.55;
   %cost[%mangoIdx] = 1.90;
   %cost[%pearIdx] = 0.68;

   %quantity[%bananaIdx] = 1; // initialize the quantity values
   %quantity[%appleIdx]  = 3;
   %quantity[%orangeIdx] = 4;
   %quantity[%mangoIdx]  = 1;
   %quantity[%pearIdx]   = 2;

   %numFruit=0;     // always a good idea to initialize *all* variables!
   %totalCost=0;    // (even if we know we are going to change them later)

   //
   // ----------------- Computation ---------------------
   //

   // Display the known statistics of the fruit collection
```

```
   for (%index = 0; %index < %numFruitTypes; %index++)
   {
      print("Cost of " @ %names[%index] @ ":$" @ %cost[%index]);
      print("Number of " @ %names[%index] @ ":" @ %quantity[%index]);
   }

   // count up all the pieces of fruit, and display that result
   for (%index = 0; %index <= %numFruitTypes; %index++)
   {
      %numFruit = %numFruit + %quantity[%index];
   }
   print("Total pieces of Fruit:" @ %numFruit);


   // now calculate the total cost
   for (%index = 0; %index <= %numFruitTypes; %index++)
   {
      %totalCost = %totalCost + (%quantity[%index]*%cost[%index]);
   }
   print("Total Price of Fruit:$" @ %totalCost);
}
```

Type this program in, save it as C:\3DGPAi1\book\FruitLoopy.cs, and then run it.

Of course, you will notice right away that I've used comments to organize the code into two sections, *initialization* and *computation*. This was purely arbitrary—but it is a good idea to label sections of code in this manner, to provide signposts, as it were. You should also notice that all the variables in the program are local, rather than global, in scope. This is more reasonable for a program of this nature, where having everything contained in one function puts all variables in the same scope.

Next you will see that I've actually created three arrays: `name`, `cost`, and `quantity`. Each array has the same number of elements, by design. Also, I have assigned appropriately named variables to carry the index values of each of the fruit types. This way I don't need to remember which fruit has which index when it comes time to initialize them with their names, prices, and counts.

Then it is just a simple matter of looping through the list to perform the operation I want.

Elegant, huh? But it could be better. See if you can find a way to reduce the number of lines of code in the computation section even more, and write your own version and try it out for yourself. I've written my own smaller version; you can find it in the C:\3DGPAi1\Book\Exercises folder, named ParedFruit.cs.

For a further illuminating exercise, try this: Rewrite FruitLoopy.cs to perform exactly the same operations, but without using arrays at all. Go ahead—take some time and give it a try. You can compare it with my version in the C:\3DGPAi1\Book\Exercises folder, named FermentedFruit.cs.

Now, the final exercise is purely up to you and your mind's eye: Imagine that you have 33 types of fruit, instead of five. Which program would you rather modify—ParedFruit.cs or FermentedFruit.cs? Can you see the advantage of arrays now?

Another thing to point out is that the initialization section of the code would probably read in the values from a database or an external file with value tables in it. It would use a loop to store all the initial values—the names, costs, and quantities. Then the code would really be a lot smaller!

To review, an array is a data structure that allows a collective name to be given to a group of elements of the same type. An individual element of an array is identified by its own unique index (or subscript).

An array can be thought of as a collection of numbered boxes, each containing one data item. The number associated with the box is the index of the item. To access a particular item, the index of the box associated with the item is used to access the appropriate box. The index must be an integer and indicates the position of the element in the array.

Strings

We've already encountered strings in our earlier example programs. In some languages strings are a special type of array, like an array of single characters, and can be treated as such. In Torque, strings are in essence the only form of variable. Numbers and text are stored as strings. They are handled as either text or numbers depending on which operators are being used on the variables.

As we've seen, two basic string operations are *assignment* and *concatenation*, as illustrated here:

```
%myFirstName = "Ken";
%myFullName = %myFirstName @ " Finney";
```

In the first line, the string `"Ken"` is assigned to `%myFirstName`, then the string `" Finney"` is concatenated (or appended) to `%myFirstName`, and the result is assigned to `%myFullName`. Familiar stuff by now, right? Well, try this one on for size:

```
%myAge = 30;                // (actually it isn't you know !)
%myAge = %myAge + 12;       // getting warmer !
```

At this point, the value in `%myAge` is 42, the sum of 30 and 12. Now watch this trick:

```
%aboutMe = "My name is " @ %myFullName @ " and I am " @ %myAge @ " years old.";
```

I'm sure you can figure out what the value of the variable `%aboutMe` is. That's right, it's one long string—"My name is Ken Finney and I am 42 years old."—with the number values embedded as text, not numbers. Of course, that isn't my age, but who's counting?

What happened is that the Torque Engine figured out by the context what operation you wanted to perform, and it converted the number to a string value before it added it to the larger string.

There is another form of string variable called the *tagged string*. This is a special string format used by Torque to reduce bandwidth utilization between the client and the server. We'll cover tagged strings in more detail in a later chapter.

Operators

Table 2.4 is a list of operators. You will find it handy to refer back to this table from time to time.

Table 2.4 Torque Script Operators

Symbol	Meaning
+	Add
−	Subtract
*	Multiply
/	Divide
%	Modulus
++	Increment by 1
--	Decrement by 1
+=	Addition totalizer
-=	Subtraction totalizer
*=	Multiplication totalizer
/=	Division totalizer
%=	Modulus totalizer
@	String append
( )	Parentheses—operator precedence promotion
[ ]	Brackets—array index delimiters
{ }	Braces—indicate start and end of code blocks
SPC	Space append macro (same as @ " " @)
TAB	Tab append macro (same as @ "\t" @)
NL	New line append (same as @ "\n" @)
~	(Bitwise NOT) Flips the bits of its operand
|	(Bitwise OR) Returns a 1 in a bit if bits of either operand is 1
&	(Bitwise AND) Returns a 1 in each bit position if bits of both operands are 1s

continued

Team LRN

^	(Bitwise XOR) Returns a 1 in a bit position if bits of one but not both operands are 1
<<	(Left-shift) Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in 0s from the right
>>	(Sign-propagating right-shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off
|=	Bitwise OR with result assigned to the first operand
&=	Bitwise AND with result assigned to the first operand
^=	Bitwise XOR with result assigned to the first operand
<<=	Left-shift with result assigned to the first operand
>>=	Sign-propagating right-shift with result assigned to the first operand
!	Evaluates the opposite of the value specified
&&	Requires both values to be true for the result to be true
||	Requires only one value to be true for the result to be true
==	Left-hand value and right-hand value are equal
!=	Left-hand value and right-hand value are not equal
<	Left-hand value is less than right-hand value
>	Left-hand value is greater than right-hand value
<=	Left-hand value is less than or equal to right-hand value
>=	Left-hand value is greater than or equal to right-hand value
$=	Left-hand string is equal to right-hand string
!$=	Left-hand string is not equal to right-hand string
//	Comment operator—ignore all text from here to the end of the line
;	Statement terminator
.	Object/data block method or property delimiter

Operators range from the familiar to the mighty weird. The familiar will be the ones like add ("+") and subtract ("−"). A little strange for those who are adept with standard secondary school math but new to programming languages is the multiplication symbol—an asterisk ("*"). The division symbol, though not the regular handwritten one, is still a somewhat familiar slash ("/"). A mighty weird one would be the vertical pipe ("|"), which is used to perform an OR operation on the bits of a variable.

Some of the operators are probably self-explanatory or understandable from the table. Others may require some explanation, which you will find in the following sections of this chapter.

You'll recall that strings and numbers are treated the same; there is, however, one exception, and that is when comparing strings to strings or numbers to numbers. We use different operators for those comparisons. For number comparisons, we use `==` (that's not a typo—it's two equal signs in a row; read it as "is identical to") and for string comparisons, we use `$=` (read it as "string is identical to"). These operators will be discussed more in the sections called "Conditional Expressions" and "Branching."

Team LRN

Operator Precedence

An issue with evaluating expressions is that of *order of evaluation*. Should `%a + %b * %c` be evaluated by performing the multiplication first or by performing the addition first? In other words, as `%a + (%b * %c)` or as `(%a + %b) * %c`?

Torque and other languages (such as C/C++) solve this problem by assigning priorities to operators; operators with high priority are evaluated before operators with low priority. Operators with equal priority are evaluated in left-to-right order. The priorities of the operators seen so far are, in order of high to low priority, as follows:

```
( )
* / %
+ -
=
```

Therefore, `%a + %b * %c` is evaluated as if it had been written as `%a + (%b * %c)` because multiplication (`*`) has a higher priority than addition (`+`). If the `+` needed to be evaluated first, then parentheses would be used as follows: `(%a + %b) * %c`.

If you have any doubt, then use extra parentheses to ensure the correct order of evaluation. Note that two arithmetic operators cannot be written in succession.

Increment/Decrement Operators

There are some operations that occur so frequently in assignment statements that Torque has shorthand methods for writing them. One common situation is that of incrementing or decrementing an integer variable. For example,

```
%n = %n + 1;  // increment by one
%n = %n - 1;  // decrement by one
```

Torque has an increment operator (`++`) and a decrement operator (`--`). Thus

```
%n++;
```

can be used for the increment and

```
%n--;
```

can be used for the decrement.

The `++` and `--` operators here have been written after the variable they affect; they are called the *postincrement* and *postdecrement operators*, respectively. Torque does not have preincrement and predecrement operators (which are written before the variable), as you would find in C/C++.

Totalizers

Totalizers are a variation on the increment and decrement theme. Instead of bumping a value up or down by 1, a totalizer does it with any arbitrary value. For example, a common situation that occurs is an assignment like this:

```
%total = %total + %more;
```

where a variable is increased by some amount and the result is assigned back to the original variable. This type of assignment can be represented in Torque by the following:

```
%total += %more;
```

This notation can be used with the other arithmetic operators (+, -, *, /, and %), as you can see in the following:

```
%prod = %prod * 10;
```

which can be written as this:

```
%prod *= 10;
```

You can use totalizers in compound assignment statements quite easily as well. Here's an example:

```
%x = %x/(%y + 1);
```

becomes

```
%x /= %y + 1;
```

and

```
%n = %n % 2;
```

becomes

```
%n %= 2;
```

Be careful on that last one! The percent sign in front of the number 2 is the modulus operator, not a scope prefix. You can tell by the space that separates it from the 2—or in the case of the totalizer example, you can tell by the fact that the percent sign is adjacent to the equal sign on the right. They are certainly subtle differences, so make sure you watch for them if you work in code that uses these constructs.

In all cases, you must be performing these operations on numbers and not strings. That wouldn't make any sense!

Team LRN

Loops

Loops are used for repetitive tasks. We saw an example of a loop being used in the FruitLoopy sample program. This loop was used to step through the available types of fruit. The loop was a *bounded* one that had a specified start and end, a characteristic built into the loop construct we used, the `for` loop. The other kind of loop we are going to look at is the `while` loop.

The while Loop

The following piece of Torque Script demonstrates a `while` loop. It gets a random number between 0 and 10 from the Torque Engine and then prints it out.

```
// ============================================================================
//  WhilingAway.cs
//
//  This module is a program that demonstrates while loops. It prints
//  random values on the screen as long as a condition is satisfied.
//
// ============================================================================

function main()
// ------------------------------------------------
//     Entry point for the program.
// ------------------------------------------------
{
   %value = 0;             // initialize %value
   while (%value < 7)     // stop looping if %n exceeds 7
   {
      %value = GetRandom(10);  // get a random number between 0 and 10
      print("value="@%value ); // print the result
   }                           // now back to the top of the loop
                               // ie. do it all again
}
```

Save this program as C:\3DGPAi1\book\WhilingAway.cs and run it. Note the output. Now run it again. Note the output again—and the fact that this time it's different. That's the randomness in action, right there. But the part that we are really interested in right now is the fact that as long as the number is less than 7, the program continues to loop.

The general form of a `while` statement is this:

```
while (  condition  )
         statement
```

While the condition is `true` the statement is executed over and over. Each time the condition is satisfied and the statement is executed is called an *iteration*. The statement may be a single statement (terminated by a semicolon) or code block (delimited by braces) when you want two or more statements to be executed. Note the following points: It must be possible to evaluate the condition on the first entry to the `while` statement or it will never be satisfied, and its code will never be executed. This means that all variables used in the condition must have been given values before the `while` statement is encountered. In the preceding example the variable `%value` was started at 0 (it was initialized) and it was given a random number between 0 and 10 during each iteration of the loop.

Now you have to make sure that at least one of the variables referenced in the condition can be changed in the statement portion that makes up the body of the loop. If you don't, you could end up stuck in an *infinite loop*. In the preceding example by making sure that the randomly chosen `%value` would always *eventually* cause the condition to fail (10 is greater than 7), we ensure that the loop will stop at some point. In fact, the random number code will return 7, 8, 9, and 10 at some point or other—any one of which will cause the code to break out of the loop.

Here is the important thing about `while` loops: The condition is evaluated *before* the loop body statements are executed. If the condition evaluates to `false` when it is first encountered, then the body is never entered. In the preceding example if we had initialized `%value` with 10, then no execution of the statements in the body of the `while` loop would have happened.

And now here's a little exercise for you. Write a program, saving it as C:\3DGPAi1\book\looprint.cs. Make the program print all the integers starting at 0 up to and including 250. That's a lot of numbers! Use a `while` loop to do it.

The for Loop

When programming, we often need to execute a statement a specific number of times. Consider the following use of a `while` statement to output the numbers 1 to 10. In this case the integer variable `i` is used to control the number of times the loop is executed.

```
%count = 1;
while (%count <= 10)
{
  print("count="@%count);
  %count++;
}
```

Three distinct operations take place:

- **Initialization.** Initializes the control variable `%count` to 1.
- **Evaluation.** Evaluates the value of an expression (`%count <= 10`).

- **Update.** Updates the value of the control variable before executing the loop again (`%count++`).

The `for` statement is specially designed for these cases—where a loop is to be executed starting from an initial value and iterates until a control condition is satisfied, meanwhile updating the value of the control variable each time around the loop. It has all three operations rolled up into its principal statement syntax. It's sort of the Swiss army knife of `loop` statements.

The general form of the `for` statement is

```
for ( initialize ; evaluate ; update )
    statement
```

which executes the initialize operation when the `for` statement is first entered. The evaluate operation is then performed on the test expression; if it evaluates to `true`, then the `loop` statement is executed for one iteration followed by the update operation. The cycle of test, iterate, update continues until the test expression evaluates to `false`; control then passes to the next statement in the program.

Functions

Functions save work. Once you've written code to solve a problem, you can roll the code into a function and reuse it whenever you encounter that problem again. You can create functions in a manner that allows you to use the code with different starting parameters and either create some effect or return a value to the code that uses the function.

When solving large problems we often use a divide-and-conquer technique, sometimes called *problem decomposition*. We break a big problem down into smaller problems that are easier to solve. This is often called the *top-down* approach. We keep doing this until problems become small enough that a single person can solve them. This top-down approach is essential if the work has to be shared among a team of programmers; each programmer ends up with a specification for a small part of the bigger system that is to be written as a function (or a collection of functions). The programmer can concentrate on the solution of only this one problem and is likely to make fewer errors. The function can then be tested on its own for correctness compared to the design specification.

There are many specialized problem areas, and not every programmer can be proficient in all of them. Many programmers working in scientific applications will frequently use math function routines like sine and cosine but would have no idea how to write the code to actually perform those operations. Likewise, a programmer working in commercial applications might know little about how an efficient sorting routine can be written. A specialist can create such routines and place them in a public library of functions, however, and all programmers can benefit from this expertise by being able to use these efficient and well-tested functions.

Team LRN

In the "Arrays" section earlier in this chapter we calculated a total price and total count of several types of fruit with the FruitLoopy program. Here is that program modified somewhat (okay, modified a *lot*) to use functions. Take note of how small the `main` function has become now that so much code is contained within the three new functions.

```
// ========================================================================
//  TwotyFruity.cs
//
//  This program adds up the costs and quantities of selected fruit types
//  and outputs the results to the display. This module is a variation
//  of the FruitLoopy.cs module designed to demonstrate how to use
//  functions.
// ========================================================================

function InitializeFruit($numFruitTypes)
// ------------------------------------------------------------------------
//     Set the starting values for our fruit arrays, and the type
//     indices
//
//     RETURNS: number of different types of fruit
//
// ------------------------------------------------------------------------
{
    $numTypes = 5; // so we know how many types are in our arrays
    $bananaIdx=0;    // initialize the values of our index variables
    $appleIdx=1;
    $orangeIdx=2;
    $mangoIdx=3;
    $pearIdx=4;

    $names[$bananaIdx] = "bananas"; // initialize the fruit name values
    $names[$appleIdx] = "apples";
    $names[$orangeIdx] = "oranges";
    $names[$mangoIdx] = "mangos";
    $names[$pearIdx] = "pears";

    $cost[$bananaIdx] = 1.15; // initialize the price values
    $cost[$appleIdx] = 0.55;
    $cost[$orangeIdx] = 0.55;
    $cost[$mangoIdx] = 1.90;
    $cost[$pearIdx] = 0.68;

    $quantity[$bananaIdx] = 1; // initialize the quantity values
```

Team LRN

```
    $quantity[$appleIdx]  = 3;
    $quantity[$orangeIdx] = 4;
    $quantity[$mangoIdx]  = 1;
    $quantity[$pearIdx]   = 2;

    return($numTypes);
}

function addEmUp($numFruitTypes)
// ---------------------------------------------------------------------
//     Add all prices of different fruit types to get a full total cost
//
//     PARAMETERS: %numTypes -the number of different fruit that are tracked
//
//     RETURNS: total cost of all fruit
//
// ---------------------------------------------------------------------
{
   %total = 0;
   for (%index = 0; %index <= $numFruitTypes; %index++)
   {
      %total = %total + ($quantity[%index]*$cost[%index]);
   }
   return %total;
}


// ---------------------------------------------------------------------
//  countEm
//
//     Add all quantities of different fruit types to get a full total
//
//     PARAMETERS: %numTypes -the number of different fruit that are tracked
//
//     RETURNS: total of all fruit types
//
// ---------------------------------------------------------------------
function countEm($numFruitTypes)
{
   %total = 0;
   for (%index = 0; %index <= $numFruitTypes; %index++)
   {
```

```
      %total = %total + $quantity[%index];
   }
   return %total;
}

function main()
// ---------------------------------------------------------------------
//     Entry point for program. This program adds up the costs
//     and quantities of selected fruit types and outputs the results to
//     the display. This program is a variation of the program FruitLoopy
//
// ---------------------------------------------------------------------
{
   //
   // ----------------- Initialization --------------------
   //

   $numFruitTypes=InitializeFruit(); // set up fruit arrays and variables
   %numFruit=0;     // always a good idea to initialize *all* variables!
   %totalCost=0;    // (even if we know we are going to change them later)

   //
   // ----------------- Computation ---------------------
   //

   // Display the known statistics of the fruit collection
   for (%index = 0; %index < $numFruitTypes; %index++)
   {
   print("Cost of " @ $names[%index] @ ":$" @ $cost[%index]);
   print("Number of " @ $names[%index] @ ":" @ $quantity[%index]);
   }

   // count up all the pieces of fruit, and display that result
   %numFruit = countEm($numFruitTypes);
   print("Total pieces of Fruit:" @ %numFruit);

   // now calculate the total cost
   %totalCost = addEmUp($numFruitTypes);
   print("Total Price of Fruit:$" @ %totalCost);
}
```

Save this program as C:\3DGPAi1\book\TwotyFruity.cs and run it in the usual way. Now go and run your FruitLoopy program, and compare the output. Hopefully, they will be exactly the same.

In this version all the array initialization has been moved out of the `main` function and into the new `InitializeFruit` function. Now, you might notice that I have changed the arrays to be global variables. The reason for this is that Torque does not handle passing arrays to functions in a graceful manner. Well, actually it does, but we would need to use `ScriptObjects`, which are not covered until a later chapter, so rather than obfuscate things too much right now, I've made the arrays into global variables. This will serve as a useful lesson in contrast between global and local variables anyway, so I thought, why not?

The global arrays can be accessed from within any function in the file. The local ones (with the percent sign prefix), however, can only be accessed within a function. This is more obvious when you look at the `addEmUp` and `countEm` functions. Notice that they both use a variable called `%total`. But they are actually two *different* variables whose scope does not extend outside the functions where they are used. So don't get mixed up!

Speaking of `addEmUp` and `countEm`, these functions have another construct, called a *parameter*. Sometimes we use the word *argument* instead, but because we are all friends here, I'll stick with parameter.

Functions with No Parameters

The function `main` has no parameters, so you can see that parameters are not always required. Because the arrays are global, they can be accessed from within any function, so we don't *need* to try to pass in the data for them anyway.

Functions with Parameters and No Return Value

Parameters are used to pass information into a function, as witnessed with the functions `addEmUp` and `countEm`. In both cases we pass a parameter that tells the function how many types of fruit there are to deal with.

The function declaration looked like this:

```
function addEmUp(%numTypes)
{
```

and when we actually used the function we did this:

```
 %totalCost = addEmUp($numFruitTypes);
```

where `$numFruitTypes` indicates how many types of fruit there are—in this case, five. This is known as a *call* to the function `addEmUp`. We could have written it as

```
 %totalCost = addEmUp(5);
```

but then we would have lost the flexibility of using the variable to hold the value for the number of fruit types.

This activity is called *parameter passing*. When a parameter is passed during a function call, the value passed into the function is assigned to the variable that is specified in the function declaration. The effect is something like `%numTypes = $numFruitTypes`; now this code doesn't actually exist anywhere, but operations are performed that have that effect. Thus, `%numTypes` (inside the function) receives the value of `$numFruitTypes` (outside the function).

Functions That Return Values

The function `InitializeFruit` returns a number for the number of different fruit types with this line:

```
return(%numTypes);
```

and the functions `addEmUp` and `countEm` both have this line:

```
return %total;
```

Notice that the first example has the variable sitting inside some parentheses, and the second example does not. Either way is valid.

Now what happens is that when Torque encounters a `return` statement in a program, it gathers up the value in the `return` statement, and then exits the function and resumes execution at the code where the function was called. There isn't always a `return` statement in a function, so don't be annoyed if you see functions without them. In the case of the `InitializeFruit` function, that would have been the line near the start of `main` that looks like this:

```
$numFruitTypes=InitializeFruit(); // set up fruit arrays and variables
```

If the function call was part of an assignment statement, as above, then whatever value was gathered at the `return` statement inside the function call is now assigned in the assignment statement. Another way of expressing this concept is to say that the function *evaluated* to the value of the `return` statement inside the function.

`Return` statements don't need to evaluate to anything, however. They can be used to simply stop execution of the function and return control to the calling program code with a return value. Both numbers and strings can be returned from a function.

Conditional Expressions

A conditional or logical expression is an expression that can only evaluate to one of two values: `true` or `false`. A simple form of logical expression is the conditional expression, which uses relational operators to construct a statement about a given condition. The following is an example of a conditional expression:

Team LRN

```
%x < %y
```

(read as `%x` is less than `%y`), which evaluates to `true` if the value of the variable `%x` is less than the value of the variable `%y`. The general form of a conditional expression is

```
operandA relational_operator operandB
```

The operands can be either variables or expressions. If an operand is an expression, then the expression is evaluated and its value used as the operand. The relational operators allowable in Torque are shown in Table 2.5.

note

Another name for logic that involves only the values `true` or `false` is *Boolean* logic.

Table 2.5 Relational Operators

Symbol	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
$=	string equal to
!$=	string not equal to

Note that equality is tested for using the operator `==` because `=` is already used for assigning values to variables. The condition evaluates to `true` if the values of the two operands satisfy the relational operator and `false` if they don't.

Here are some examples:

```
%i < 10
%voltage >= 0.0
%total < 1000.0
%count != %n
%x * %x + %y * %y < %r * %r
```

Depending on the values of the variables involved, each of the preceding expressions is `true` or `false`. If `%x` has the value 3, `%y` is 6, and `%r` is 10, the last expression evaluates to `true`, but if `%x` was 7 and `%y` was 8, then it would evaluate to `false`.

The value of a logical expression can be stored in a variable for later use. Any numerical expression can be used for the value of a condition, with 0 being interpreted as `false` and 1 as `true`.

This means that the value a logical expression evaluates to can be used in arithmetical operations. This is often done by programmers, but it is a practice not to be recommended. It can lead to code obscurity, creating a program that is difficult to understand.

Logical Expressions

We can create more complex conditions than those that can be written using only the relational operators described in the preceding section. There are explicit logical operators for combining the logical values `true` and `false`.

The simplest logical operator is NOT, which is represented in Torque by the exclamation point ("!"). It operates on a single operand and returns `false` if its operand is `true` and `true` if its operand is `false`.

The operator AND, represented by two ampersands ("&&"), takes two operands and is `true` only if both of the operands are `true`. If either operand is `false`, the resulting value is `false`.

The final logical operator is OR, which is represented by two vertical pipes ("||"). It results in `true` if either operand is `true`. It returns `false` only if both its operands are `false`.

The logical operators can be defined by truth tables as seen in Table 2.6. The "F" character is used for `false` and "T" is used for `true` in these tables.

Table 2.6 Logical Operator Truth Tables

NOT (!)		OR (||)			AND (&&)		
A	**!A**	**A**	**B**	**A OR B**	**A**	**B**	**A AND B**
F	T	T	T	T	T	T	T
T	F	T	F	T	T	F	F
		F	T	T	F	T	F
		F	F	F	F	F	F

These tables show that NOT reverses the truth value of the operand A; that the AND of two operands is only `true` if both operands are `true`; and that the OR of two operands is `true` if either or both of its operands are `true`. Now we can write pretty complex logical operations.

If `%i` has the value 15, and `%j` has the value 10, then the expression `(i > 10) && (j > 0)` is evaluated by evaluating the relation `i > 10` (which is `true`), then evaluating the relation `%j > 0` (which is also `true`), to give `true`. If `%j` has the value -1, then the second relation would

be `false`, so the overall expression would be `false`. If `i` has the value 5, then the first relation would be `false`, and the expression will be `false` irrespective of the value of the second relation. Torque does not even evaluate the second relation in this situation. Similarly, if the first relation is `true` in an OR (`||`) expression, then the second relation will not be evaluated. This short-circuit evaluation enables many logical expressions to be efficiently evaluated.

Examples Using Logical Operators

Note that in the last example that follows, an actual truth value (0 or `false`) was used as one of the operands of `&&`. This means that whatever the value of `%i`, this logical expression evaluates to `false`. In these examples parentheses have been used to clarify the order of operator application.

```
(%i < 10)  &&  (%j > 0)
((%x + %y) <= 15) || (%i == 5)
!((%i >= 10) || (%j <= 0))
(%i < 10) && 0
```

You've got to be careful not to confuse the assignment operator = with the logical equality operator ==. Using Table 2.6 with the following expression

```
x + y < 10 && x/y == 3 || z != 10
```

shows that the operators are evaluated in the order /, +, <, ==, !=, &&, and ||. This is the same as using parentheses on the expression in this way: `((((x + y) < 10) && ((x/y) == 3)) || (z != 10))`.

Similarly, the expressions given above could be written without parentheses as follows:

```
i < 10  &&  j > 0
x + y <= 15 || i == 5
!(i >= 10 || j <= 0)
i < 10 && 0
```

Now that we've covered the logical expressions (or conditions) in Torque, let's move on and take a look at the conditional control mechanisms in Torque.

Branching

The term *branching* refers to the idea that code can follow different execution paths depending on, well, something. What it depends on…ummm…depends. Well, let me try that again. It depends on what your program is doing and what you want it to do. Like this: Say you are driving on a road, and you reach a T junction. The sign points left and says "Toronto 50 km." Another sign points right and says "Toronto (Scenic Route) 150 km." Which way are you going to go, left or right? Well, you see? It depends. The fastest way to Toronto might be to go left, but what if you aren't in a hurry—maybe you're

interested in the scenic route? Just as we've seen earlier with looping, there are conditions that will dictate what path your code will take.

That act of taking one path over others available is branching. Branching starts out with some sort of decision-making test. In addition to the two looping statements we've already covered—which employ branching of sorts—there are also two branch-specific statements: the `if` statement and the `switch` statement.

The if Statement

The simplest way to select the next thing to do in a program based upon conditions is to use the `if` statement. Check this out:

```
if (%n > 0)
   print("n is a positive number");
```

This will print out the message `n is a positive number` only if `%n` is positive. The general form of the `if` statement is this:

```
if (condition)
   statement
```

where `condition` is any valid logical expression as described in the "Conditional Expressions" section we saw earlier.

This `if` statement adds `%something` to the variable `%sum` if `%something` is positive:

```
if (%something > 0)
  %sum += %something;
```

If `%something` isn't positive, then the program branches *past* the totalizer statement, and so `%sum` doesn't get incremented by `%something`.

This next piece of code also adds `%something` to `%sum`, but it also increments a positive number counter called `%poscount`:

```
if (%something > 0)
{
   %sum += %something;
   %counter++;
}
```

Note how in the second example a compound statement has been used to carry out more than one operation if the condition is `true`. If it had been written like this:

```
if (%something > 0)
  %sum += %something;
  %counter++;
```

then if %something was greater than 0 the next statement would be executed—that is, %sum would incremented by the amount of %something. But the statement incrementing %counter is now going to be treated as the next statement in the program and not as part of the if statement. The program execution is not going to branch around it. The effect of this would be that %counter would be incremented every time it is encountered, no matter whether %something is positive or negative.

The statements within a compound statement can be any Torque statements. In fact, another if statement could be included. For example, the following code will print a message if a quantity is negative and a further message if no overdraft has been arranged:

```
if ( %balance < 0 )
{
   print ("Your account is overdrawn. Balance is: " @ %balance );
   if ( %overdraft <= 0 )
      print ("You have exceeded your overdraft limit");
}
```

Now we could have done the same thing using two sequential if statements and more complex conditions:

```
if ( %balance < 0 )
   print ("Your account is overdrawn. Balance is: " @ %balance );
if ( %balance < 0 && %overdraft <= 0 )
      print ("You have exceeded your overdraft limit");
```

You should note that one of these versions will generally execute a little bit faster than the second when dealing with accounts that are not overdrawn. Before I tell you later in this chapter, see if you can figure out which one, and why.

The if-else Statement

A simple if statement only allows a single branch to a simple or compound statement when a condition holds. Sometimes there are alternative paths, some that need to be executed when the condition holds, and some to be executed when the condition does not hold. The two forms can be written this way:

```
if (%coffeeholic == true)
      print ("I like coffee.");
if (%coffeeholic == false)
      print ("I don't like coffee.");
```

This technique will work while the statements that are executed as a result of the first comparison do not alter the conditions under which the second if statement are executed. Torque provides a direct means of expressing these kinds of choices. The if-else statement specifies statements to be executed for both possible logical values of the condition in an

if statement. The following example of an if-else statement writes out one message if the variable %coffeeholic is positive and another message if %coffeeholic is negative:

```
if (%coffeeholic == true)
       print ("I like coffee.");
else
       print ("I don't like coffee.");
```

The general form of the if-else statement is this:

```
if ( condition )
   statementA
else
   statementB
```

If the condition is true, then statementA is executed; otherwise statementB is executed. Both statementA and statementB may be either simple or compound statements.

The following if-else statement evaluates if a fruit is fresh or not, and if it is, the statement increments a fresh fruit counter. If the fruit isn't fresh, the statement increments the rotten fruit counter. I'm going to program my refrigerator's fruit crisper to do this one day and send me reports over the Internet. Well, I can wish, can't I?

```
if (%fruitState $= "fresh")
{
    %freshFruitCounter++;
}
else
{
    %rottenFruitCounter++;
}
```

Time for another sample program! Type the following program in and save it as C:\3DGPAi1\book\Geometry.cs and then run it.

```
// ============================================================================
//  geometry.cs
//
//  This program calculates the distance around the perimeter of
//  a quadrilateral as well as the area of the quadrilateral and outputs the
//  values. It recognizes whether the quadrilateral is a square or a rectangle and
//  modifies its output accordingly. Program assumes that all angles in the
//  quadrilateral are equal. Demonstrates the if-else statement.
// ============================================================================

function calcAndPrint(%theWidth, %theHeight)
```

```
// ------------------------------------------------------------------------
//    This function does the shape analysis and prints the result.
//
//    PARAMETERS: %theWidth - horizontal dimension
//                %theHeight - vertical dimension
//
//    RETURNS: none
// ------------------------------------------------------------------------
{
  // calculate perimeter
  %perimeter = 2 * (%theWidth+%theHeight);

  // calculate area
  %area = %theWidth * %theHeight;

  // first, set up the dimension output string
  %prompt = "For a " @ %theWidth @ " by " @
             %theHeight @ " quadrilateral, area and perimeter of ";

  // analyze the shape's dimensions and select different
  // descriptors based on the shape's dimensions
  if (%theWidth == %theHeight)                // if true, then it's a square
    %prompt = %prompt @ "square: ";
  else                                          // otherwise it's a rectangle
    %prompt = %prompt @ "rectangle: ";

  // always output the analysis
  print (%prompt @ %area @ " " @ %perimeter);
}

function main()
// ------------------------------------------------------------------------
//     Entry point for the program.
// ------------------------------------------------------------------------
{

   // calculate and output the results for three
   // known dimension sets
   calcAndPrint(22, 26); // rectangle
   calcAndPrint(31, 31); // square
   calcAndPrint(47, 98); // rectangle
}
```

Team LRN

What we've done here is analyze a shape. In addition to printing its calculated measurements, we modify our output string based upon the (simple) analysis that determines if it is a square or a rectangle. I realize that a square *is* a rectangle, but let's not get too picky, okay? Not yet, at least.

Nesting if Statements

You saw earlier in "The `if` Statement" section how an `if` statement can contain another `if` statement. These are called *nested `if` statements.* There is no real limit to how deep you can nest the statements, but try to be reasonable and only do it if it is absolutely necessary for functional reasons. It might be good to do it for performance reasons, and that's fine as well.

By the way, I had asked if you could tell which of the two examples would execute faster, remember that? The answer is that the nested version will execute faster when there is no overdraft condition. This is because only one condition is tested, resulting in less work for the computer to do. The sequential version will always perform both tests, no matter what the bank balance is.

The `if` and `if-else` statements allow a choice to be made between two possible alternatives. Well, sometimes we need to choose between more than two alternatives. For example, the following `sign` function returns −1 if the argument is less than 0, returns +1 if the argument is greater than 0, and returns 0 if the argument is 0.

```
function sign (%value)
//   determines the arithmetic sign of a value
//
//   PARAMETERS: %value - the value to be analyzed
//
//   RETURNS: -1   - if value is negative
//             0   - if value is zero
//             1   - if value is positive
{
   if (%value < 0) // is it negative ?
   {
      return -1;
   }
   else             // nope, not negative
   {
      if (%value == 0) // is it zero ?
      {
         return 0;
      }
      else           // nope, then it must be positive
      {
```

```
            return 1;
        }
    }
}
```

So there you go. The function has an `if-else` statement in which the statement following the `else` is also an `if-else` statement. If `%value` is less than 0, then `sign` returns −1, but if it is not less than 0, the statement following the `else` is executed. In that case if `%value` is equal to 0, then `sign` returns 0; otherwise it returns 1. I used the compound statement form in order to make the nesting stand out more. The nesting could also be written like this:

```
if (%value < 0) // is it negative ?
   return -1;
else             // nope, not negative
   if (%value == 0) // is it zero ?
      return 0;
   else           // nope, then it must be positive
      return 1;
```

This is nice and compact, but it can sometimes be hard to discern where the nesting properly happens, and it is easier to make mistakes. Using the compound form formalizes the nesting a bit more, and personally, I find it more readable.

Newbie programmers sometimes use a sequence of `if` statements rather than nested `if-else` statements when the latter should be used. They would write the guts of the `sign` function like this:

```
if (%value < 0)
   %result = -1;
if (%value == 0)
   %result = 0;
if (%value > 0)
   %result = 1;
   return %result;
```

It would work and it's fairly easy to read, but it's inefficient because all three conditions are always tested.

If nesting is carried out to too deep a level and indenting is not consistent, then deeply nested `if` or `if-else` statements will be confusing to read and interpret. You should note that an `else` always belongs to the closest `if` without an `else`.

The switch Statement

We just explored how we can choose between more than two possibilities by using nested `if-else` statements. There is a sleeker and more readable method available for certain kinds of multiple choice situations—the `switch` statement. For example, the following `switch` statement will set a game's weapon label based upon a numeric weapon type variable:

```
switch (%weaponType)
{
    case 1:  %weaponName = "knife";
    case 2:  %weaponName = "pistol";
    case 3:  %weaponName = "shotgun";
    case 4:  %weaponName = "bfg1000";
    default: %weaponName = "fist";
}
```

Here is what that would look like using `if-else`:

```
if (%weaponType == 1)
   %weaponName = "knife";
else if (%weaponType == 2)
   %weaponName = "pistol";
else if (%weaponType == 3)
   %weaponName = "shotgun";
else if (%weaponType == 4)
   %weaponName = "bfg1000";
else
   %weaponName = "fist";
```

It's pretty obvious from that simple example why the `switch` statement is so useful.

The general form of a `switch` statement is this:

```
switch ( selection-variable )
{
      case label1:
                  statement1;
      case label2:
                  statement2;
          ...
      case labeln:
                  statementn;
      default:
                  statementd;
}
```

The selection-variable may be a number or a string or an expression that evaluates to a number or a string. The selection-variable is evaluated and compared with each of the `case` labels. The `case` labels all have to be different. If a match is found between the selection-variable and one of the `case` labels, then the statements that follow the matched `case` until the next `case` statement will be executed. If the value of the selection-variable can't be matched with any of the `case` labels, then the statements associated with `default` are executed. The `default` is not required but should only be left out if it is certain that the selection-variable will always take the value of one of the `case` labels.

Here is another example, which writes out the day of the week depending on the value of the number variable `%day`.

```
switch (%day)
{
   case 1 :
               print("Sunday");
   case 2 :
               print("Monday");
   case 3 :
               print("Tuesday");
   case 4 :
               print("Wednesday");
   case 5 :
               print("Thursday");
   case 6 :
               print("Friday");
   case 7 :
               print("Saturday");
  default :
               print("Not a valid day number");
}
```

Debugging and Problem Solving

When you run your programs, the Torque Engine will automatically compile them and output a new .cs.dso file if it needs to. Therefore, geometry.cs (the source code) will become geometry.cs.dso (the compiled code). There is a gotcha though: If the script compiler detects an error in your code, it will abort the compilation, but will not stop the program execution—rather, *it will use the existing compiled version if one exists.* This is an important point to remember. If you are changing your code, yet you don't see any change in behavior, then you should check the log file in console.log and look for any compile errors.

The log output is pretty verbose and should guide you to the problem area pretty quickly. It writes out a piece of code around the problem area and then inserts a pair of sharp characters ("##") on either side of the exact spot where the compiler thinks there is a problem.

Once you've fixed the first problem, don't assume you are done. Quite often, once one problem is fixed, the compiler marches on through the code and finds another one. The compiler always aborts as soon as it encounters the first problem.

Of the large number of programming errors that the compiler catches and identifies, here are a few specific ones that frequently crop up:

- Missing semicolon at the end of a statement.
- Missing a slash in double-slash comment operator.
- Missing `%` or `$` (scope prefix) from variable names.
- Using uninitialized variables.
- Mixing global and local scope prefixes.
- Unbalanced parentheses or braces.

In a later chapter we will cover how to use the console mode in Torque. That will give us access to three built-in Torque functions—`echo`, `warn`, and `error`—which are quite useful for debugging.

Without using those three functions, the best tool for debugging programs you've created is the `print` statement. You should print out interim results throughout your code that will tell you how your program is progressing.

Tell you what—here is a different version of the TwotyFruity program. Type it in and save it as C:\3DGPAi1\book\WormyFruit.cs. I've put five bugs in this version. See if you can spot them (in addition to any you might introduce while typing).

```
// ============================================================================
//  WormyFruit.cs
//
//  Buggy version of TwotyFruity. It has five known bugs in it.
//  This program adds up the costs and quantities of selected fruit types
//  and outputs the results to the display. This module is a variation
//  of the FruitLoopy.cs module designed to demonstrate how to use
//  functions.
// ============================================================================

function InitializeFruit(%numFruitTypes)
// ----------------------------------------------------------------------------
//     Set the starting values for our fruit arrays, and the type
//     indices
```

```
//
//     RETURNS: number of different types of fruit
//
// ---------------------------------------------------------------------
{
    $numTypes = 5; // so we know how many types are in our arrays
    $bananaIdx=0;    // initialize the values of our index variables
    $appleIdx=1;
    $orangeIdx=2;
    $mangoIdx=3;
    $pearIdx=3;

    $names[$bananaIdx] = "bananas"; // initialize the fruit name values
    $names[$appleIdx] = "apples";
    $names[$orangeIdx] = "oranges";
    $names[$mangoIdx] = "mangos";
    $names[$pearIdx] = "pears";

    $cost[$bananaIdx] = 1.15; // initialize the price values
    $cost[$appleIdx] = 0.55;
    $cost[$orangeIdx] = 0.55;
    $cost[$mangoIdx] = 1.90;
    $cost[$pearIdx] = 0.68;

    $quantity[$bananaIdx] = 1; // initialize the quantity values
    $quantity[$appleIdx]  = 3;
    $quantity[$orangeIdx] = 4;
    $quantity[$mangoIdx]  = 1;
    $quantity[$pearIdx]   = 2;

    return(%numTypes);
}

function addEmUp($numFruitTypes)
// ---------------------------------------------------------------------
//     Add all prices of different fruit types to get a full total cost
//
//     PARAMETERS: %numTypes -the number of different fruit that are tracked
//
//     RETURNS: total cost of all fruit
//
// ---------------------------------------------------------------------
```

```
{
   %total = 0;
   for (%index = 0; %index <= $numFruitTypes; %index++)
   {
      %total = %total + ($quantity[%index]*$cost[%index]);
   }
   return %total;
}


// ---------------------------------------------------------------------
//  countEm
//
//     Add all quantities of different fruit types to get a full total
//
//     PARAMETERS: %numTypes -the number of different fruit that are tracked
//
//     RETURNS: total of all fruit types
//
// ---------------------------------------------------------------------
function countEm($numFruitTypes)
{
   %total = 0;
   for (%index = 0; %index <= $numFruitTypes; %index++)
   {
      %total = %total + $quantity[%index];
   }
}

function main()
// ---------------------------------------------------------------------
//     Entry point for program. This program adds up the costs
//     and quantities of selected fruit types and outputs the results to
//     the display. This program is a variation of the program FruitLoopy
//
// ---------------------------------------------------------------------
{
   //
   // ----------------- Initialization --------------------
   //

   $numFruitTypes=InitializeFruit(); // set up fruit arrays and variables
```

```
   %numFruit=0     // always a good idea to initialize *all* variables!
   %totalCost=0;    // (even if we know we are going to change them later)

   //
   // ----------------- Computation --------------------
   //

   // Display the known statistics of the fruit collection
   for (%index = 0; %index < $numFruitTypes; %index++)
   {
   print("Cost of " @ $names[%index] @ ":$" @ $cost[%index]);
   print("Number of " @ $names[%index] @ ":" @ $quantity[%index]);
   }

   // count up all the pieces of fruit, and display that result
   %numFruit = countEm($numFruitTypes));
   print("Total pieces of Fruit:" @ %numFruit);

   // now calculate the total cost
   %totalCost = addEmUp($numFruitTypes);
   print("Total Price of Fruit:$" @ %totalCost);
}
```

Run the program, and use the original TwotyFruity output as a specification to tell you whether or not this program is working correctly.

Best Practices

Programming is as much an art as it is anything else. There are often quite strenuous discussions between programmers about the best way to do certain things. However, there is consensus on a few practices that are considered to be good.

So take the following list as a guideline, and develop a style that is comfortable for you.

- Use module and function header comments to document your code.
- Sprinkle lots of commentary through your code, and make sure that it actually explains what is happening.
- Don't comment obvious things. Save the effort for the stuff that matters.
- Use white space (blank lines and spaces) to improve readability.
- Indent your code with readability in mind.
- Decompose large problems into small ones, and assault the small problems with functions.

Team LRN

- Organize your code into separate modules, and make sure the module file name is appropriate for the content, and vice versa.
- Restrict the number of lines of code you put in a module. Pick a size that suits you—about 1,000 lines should be near your upper limit.
- Use descriptive and meaningful variable names.
- While keeping your variable names descriptive, don't let the names get too long.
- Never embed tabs in code—use spaces instead. When you view your code later, you may have different tab settings, and therefore find the code hard to read. Using spaces guarantees that the visual appearance is consistent. Three spaces for an indent is a good number.
- Be consistent in your programming style decisions.
- Be alert to what programming decisions you make that work well for you, and try to consistently employ those techniques.
- Keep a change log of your work so you can keep track of the evolution of your programs.
- Use revision control software to manage your program versions.

Moving Right Along

You've now bitten off a fairly big chunk o' stuff. You've learned a new tool—in fact, a new *kind* of tool—the programmer's editor. After getting a handle on UltraEdit-32, we looked at how software does its thing, bringing people and computer hardware together by using programming languages.

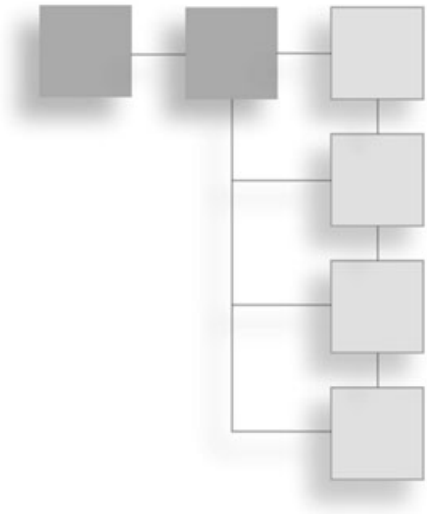
We then went off and started bullying the computer around, using one of those programming languages called Torque Script.

Coming up next, we'll delve into the world of 3D programming at a similar level, and discover the basics of 3D objects, and then how we can manipulate them with Torque Script.

This page intentionally left blank

Team LRN

CHAPTER 3



3D Programming Concepts

In this chapter we will discuss how objects are described in their three dimensions in different 3D coordinate systems, as well as how we convert them for use in the 2D coordinate system of a computer display. There is some math involved here, but don't worry—I'll do the heavy lifting.

We'll also cover the stages and some of the components of the rendering pipeline—a conceptual way of thinking of the steps involved in converting an abstract mathematical model of an object into a beautiful on-screen picture.

3D Concepts

In the real world around us, we perceive objects to have measurements in three directions, or dimensions. Typically we say they have height, width, and depth. When we want to represent an object on a computer screen, we need to account for the fact that the person viewing the object is limited to perceiving only two actual dimensions: height, from the top toward the bottom of the screen, and width, across the screen from left to right.

note

Remember that we will be using the Torque Game Engine to do most of the rendering work involved in creating our game with this book. However, a solid understanding of the technology described in this section will help guide you in your decision-making later on when you will be designing and building your own models or writing code to manipulate those models in real time.

Therefore, it's necessary to simulate the third dimension, depth "into" the screen. We call this on-screen three-dimensional (3D) simulation of a real (or imagined) object a *3D model*. In order to make the model more visually realistic, we add visual characteristics,

Team LRN

such as shading, shadows, and textures. The entire process of calculating the appearance of the 3D model—converting it to an entity that can be drawn on a two-dimensional (2D) screen and then actually displaying the resulting image—is called *rendering*.

Coordinate Systems

When we refer to the dimensional measurement of an object, we use number groups called *coordinates* to mark each *vertex* (corner) of the object. We commonly use the variable names X, Y, and Z to represent each of the three dimensions in each coordinate group, or triplet. There are different ways to organize the meaning of the coordinates, known as *coordinate systems*.

We have to decide which of our variables will represent which dimension—height, width, or depth—and in what order we intend to reference them. Then we need to decide where the zero point is for these dimensions and what it means in relation to our object. Once we have done all that, we will have defined our coordinate system.

When we think about 3D objects, each of the directions is represented by an *axis*, the infinitely long line of a dimension that passes through the zero point. Width or left-right is usually the X-axis, height or up-down is usually the Y-axis, and depth or near-far is usually the Z-axis. Using these constructs, we have ourselves a nice tidy little *XYZ-axis system*, as shown in Figure 3.1.

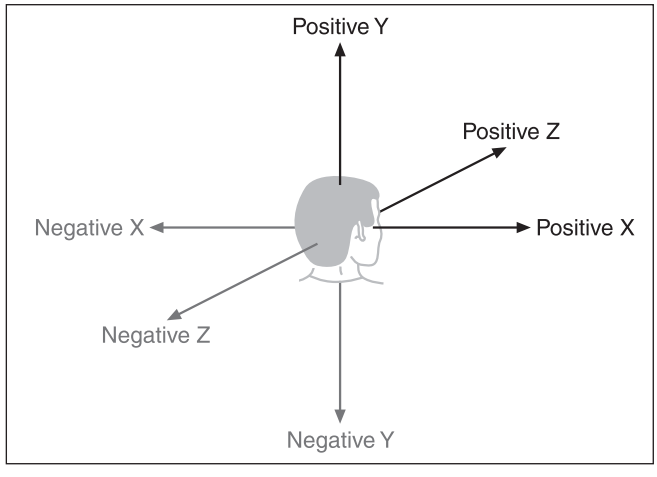


Figure 3.1 XYZ-axis system.

Now, when we consider a single object in isolation, the 3D space it occupies is called *object space*. The point in object space where X, Y, and Z are all 0 is normally the *geometric center* of an object. The geometric center of an object is usually inside the object. If positive X values are to the right, positive Y values are up, and positive Z values are away from you, then as you can see in Figure 3.2, the coordinate system is called *left-handed*.

The Torque Game Engine uses a slightly different coordinate system, a *right-handed* one. In this system, with Y and Z oriented the same as we saw in the left-handed system, X is positive in the opposite direction. In what some people call *Computer Graphics Aerobics*, we can use the thumb, index finger, and middle finger of our hands to easily figure out the handedness of the system we are using (see Figure 3.3). Just remember that using this

technique, the thumb is always the Y-axis, the index finger is the Z-axis, and the middle finger is the X-axis.

With Torque, we also orient the system in a slightly different way: The Z-axis is up-down, the X-axis is somewhat left-right, and the Y-axis is somewhat near-far (see Figure 3.4). Actually, *somewhat* means that we specify left and right in terms of looking down on a map from above, with north at the top of the map. Right and left (positive and negative X) are east and west, respectively, and it follows that positive Y refers to north and negative Y to south. Don't forget that positive Z would be up, and negative Z would be down. This is a right-handed system that orients the axes to align with the way we would look at the world using a map from above. By specifying that the zero point for all three axes is a specific location on the map, and by using the coordinate system with the orientation just described, we have defined our *world space*.

Now that we have a coordinate system, we can specify any location on an object or in a world using a coordinate triplet, such as (5,−3,−2) (see Figure 3.5). By convention, this would be interpreted as X=5, Y=−3, Z=−2. A 3D triplet is always specified in XYZ format.

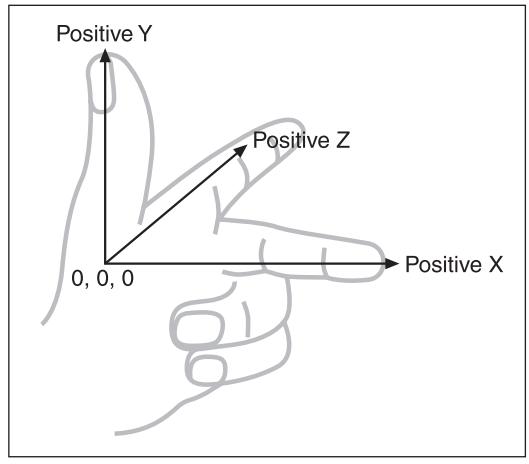


Figure 3.2 Left-handed coordinate system with vertical Y-axis.

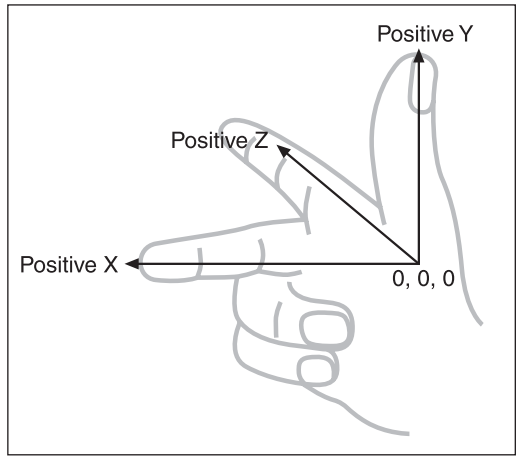


Figure 3.3 Right-handed coordinate system with vertical Y-axis.

Take another peek at Figure 3.5. Notice anything? That's right—the Y-axis is vertical with the positive values above the 0, and the Z-axis positive side is toward us. It is still a right-handed coordinate system. The right-handed system with *Y-up* orientation is often used for modeling objects in isolation, and of course we call it *object space*, as described earlier. We are going to be working with this orientation and coordinate system for the next little while.

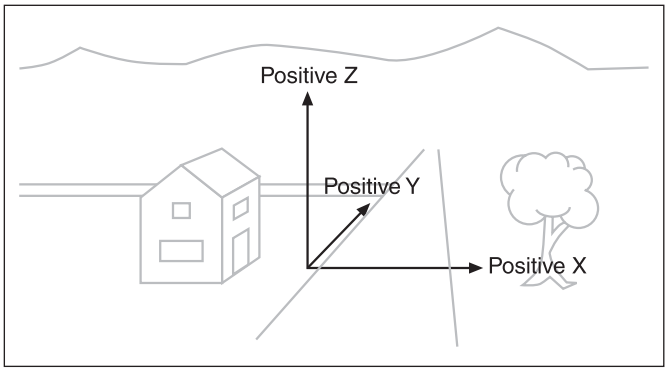


Figure 3.4 Right-handed coordinate system with vertical Z-axis depicting world space.

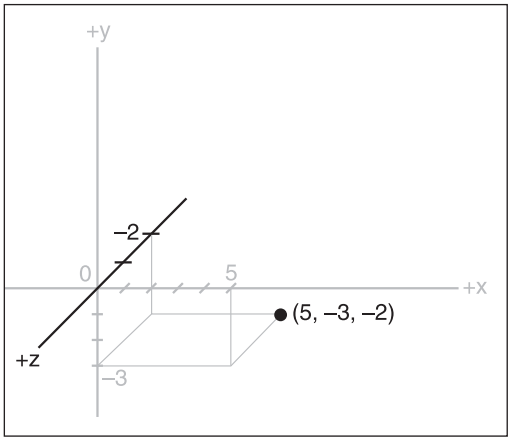


Figure 3.5 A point specified using an XYZ coordinate triplet.

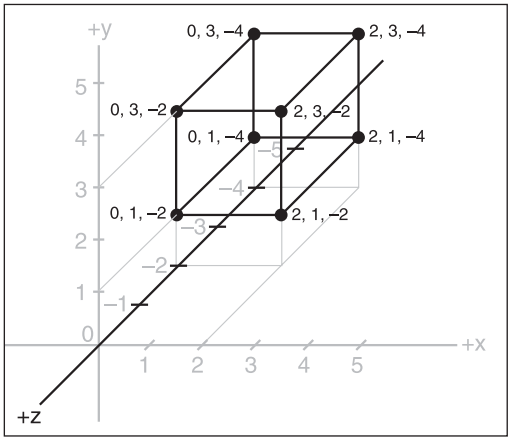


Figure 3.6 Simple cube shown in a standard XYZ axis chart.

3D Models

I had briefly touched on the idea that we can simulate, or model, any object by defining its shape in terms of its significant *vertices* (plural for *vertex*). Let's take a closer look, by starting with a simple 3D shape, or *primitive*—the cube—as depicted in Figure 3.6.

The cube's dimensions are two units wide by two units deep by two units high, or 2×2×2. In this drawing, shown in object space, the geometric center is offset to a position outside the cube. I've done this in order to make it clearer what is happening in the drawing, despite my statement earlier that geometric centers are usually located inside an object. There are times when exceptions are not only possible, but necessary—as in this case.

Examining the drawing, we can see the object's shape and its dimensions quite clearly. The lower-left-front corner of the cube is located at the position where X=0, Y=1, and Z=−2. As an exercise, take some time to locate all of the other vertices (corners) of the cube, and note their coordinates.

If you haven't already noticed on your own, there is more information in the drawing than actually needed. Can you see how we can plot the coordinates by using the guidelines to find the positions on the axes of the vertices? But we can also see the actual coordinates of the vertices drawn right in the chart. We don't need to do both. The axis lines with their index tick marks and values really clutter up the drawing, so it has become somewhat accepted in computer graphics to not

bother with these indices. Instead we try to use the minimum amount of information necessary to completely depict the object.

We only really need to state whether the object is in object space or world space and indicate the raw coordinates of each vertex. We should also connect the vertices with lines that indicate the edges.

If you take a look at Figure 3.7 you will see how easy it is to extract the sense of the shape, compared to the drawing in Figure 3.6. We specify which space definition we are using by the small XYZ-axis notation. The color code indicates the axis name, and the axis lines are drawn only for the positive directions. Different modeling tools use different color codes, but in this book dark yellow (shown as light gray) is the X-axis, dark cyan (medium gray) is the Y-axis, and dark magenta (dark gray) is the Z-axis. It is also common practice to place the XYZ-axis key at the geometric center of the model.

Figure 3.8 shows our cube with the geometric center placed where it reasonably belongs when dealing with an object in object space.

Now take a look at Figure 3.9. It is obviously somewhat more complex than our simple cube, but you are now armed with everything you need to know in order to understand it. It is a screen shot of a four-view drawing from the popular shareware modeling tool MilkShape 3D, in which a 3D model of a soccer ball was created.

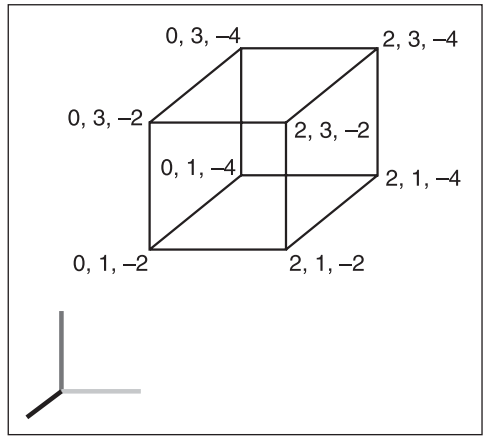


Figure 3.7 Simple cube with reduced XYZ-axis key.

In the figure, the vertices are marked with red dots (which show as black in the picture), and the edges are marked with light gray lines. The axis keys are visible, although barely so in some views because they are obscured by the edge lines. Notice the grid lines that are used to help with aligning parts of the model. The three views with the gray background and grid lines are 2D construction views, while the fourth view, in the lower-right corner, is a 3D projection of the object. The upper-left view looks down from above, with the Y-axis in the vertical direction and the X-axis in the horizontal direction. The Z-axis in that view is not visible. The upper-right view is looking at the object from the front, with the Y-axis vertical

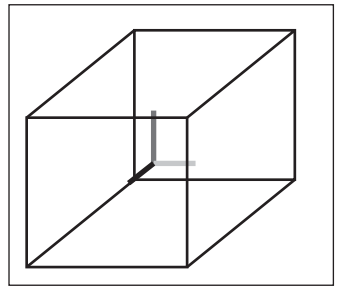


Figure 3.8 Simple cube with axis key at geometric center.

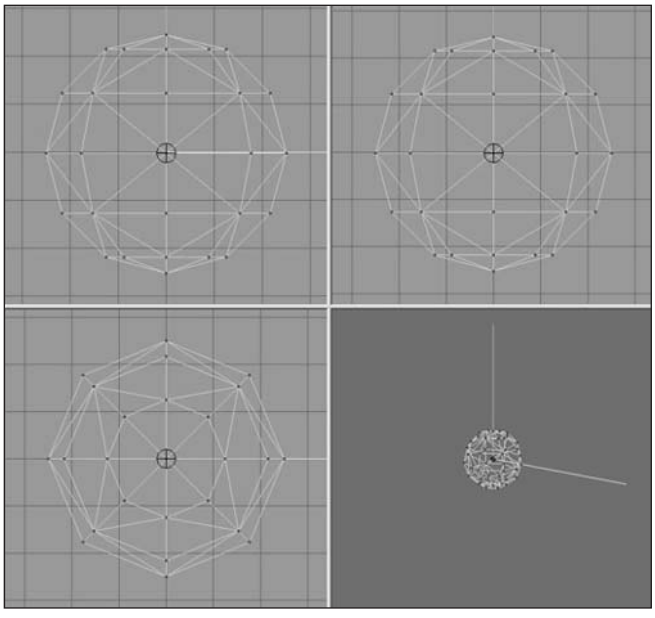


Figure 3.9 Screen shot of sphere model.

and the Z-axis horizontal; there is no X-axis. The lower-left view shows the Z-axis vertically and the X-axis horizontally with no Y-axis. In the lower-right view, the axis key is quite evident, as its lines protrude from the model.

3D Shapes

We've already encountered some of things that make up 3D models. Now it's time to round out that knowledge.

As we've seen, vertices define the shape of a 3D model. We connect the vertices with lines known as *edges*. If we connect three or more vertices with edges to create a closed figure, we've created a *polygon*. The simplest polygon is a triangle. In modern 3D accelerated graphics adapters, the hardware is designed to manipulate and display millions and millions of triangles in a second. Because of this capability in the adapters, we normally construct our models out of the simple triangle polygons instead of the more complex polygons, such as rectangles or pentagons (see Figure 3.10).

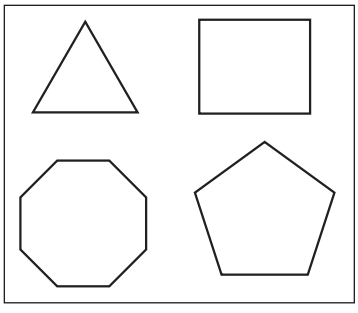


Figure 3.10 Polygons of varying complexity.

By happy coincidence, triangles are more than up to the task of modeling complex 3D shapes. Any complex polygon can be decomposed into a collection of triangles, commonly called a *mesh* (see Figure 3.11).

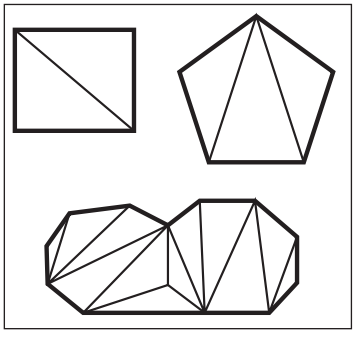


Figure 3.11 Polygons decomposed into triangle meshes.

The area of the model is known as the *surface*. The polygonal surfaces are called facets—or at least that is the traditional name. These days, they are more commonly called faces. Sometimes a surface can only be viewed from one side, so when you are looking at it from its "invisible" side, it's called a *hidden surface*, or *hidden face*. A *double-sided face* can be viewed from either side. The edges of hidden surfaces are called *hidden lines*. With

most models, there are faces on the back side of the model, facing away from us, called *backfaces* (see Figure 3.12). As mentioned, most of the time when we talk about faces in game development, we are talking about triangles, sometimes shortened to *tris*.

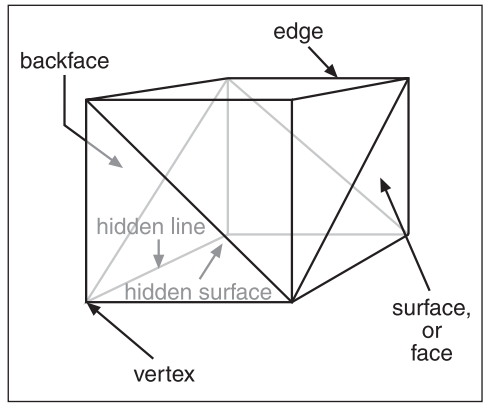


Figure 3.12 The parts of a 3D shape.

Displaying 3D Models

After we have defined a model of a 3D object of interest, we may want to display a view of it. The models are created in object space, but to display them in the 3D world, we need to convert them to world space coordinates. This requires three conversion steps beyond the actual creation of the model in object space.

1. Convert to world space coordinates.
2. Convert to view coordinates.
3. Convert to screen coordinates.

Each of these conversions involves mathematical operations performed on the object's vertices.

The first step is accomplished by the process called *transformation*. Step 2 is what we call *3D rendering*. Step 3 describes what is known as *2D rendering*. First we will examine what the steps do for us, before getting into the gritty details.

Transformation

This first conversion, to world space coordinates, is necessary because we have to place our object somewhere! We call this conversion *transformation*. We will indicate where by applying transformations to the object: a scale operation (which controls the object's size), a *rotation* (which sets orientation), and a *translation* (which sets location).

World space transformations assume that the object starts with a transformation of (1.0,1.0,1.0) for scaling, (0,0,0) for rotation, and (0,0,0) for translation.

Every object in a 3D world can have its own 3D transformation values, often simply called *transforms*, that will be applied when the world is being prepared for rendering.

tip

Other terms used for these kinds of XYZ coordinates in world space are *Cartesian coordinates*, or *rectangular coordinates*.

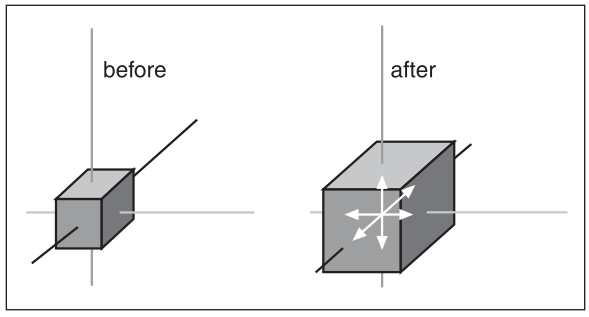


Figure 3.13 Scaling.

Scaling

We scale objects based upon a triplet of scale factors where 1.0 indicates a scale of 1:1.

The scale operation is written similarly to the XYZ coordinates that are used to denote the transformation, except that the scale operation shows how the size of the object has changed. Values greater than 1.0 indicate that the object will be made larger, and values less than 1.0 (but greater than 0) indicate that the object will shrink.

For example, 2.0 will double a given dimension, 0.5 will halve it, and a value of 1.0 means no change. Figure 3.13 shows a scale operation performed on a cube in object space. The original scale values are (1.0,1.0,1.0). After scaling, the cube is 1.6 times larger in all three dimensions, and the values are (1.6,1.6,1.6).

Rotation

The rotation is written in the same way that XYZ coordinates are used to denote the transformation, except that the rotation shows how much the object is rotated around each of its three axes. In this book, rotations will be specified using a triplet of degrees as the unit of measure. In other contexts, radians might be the unit of measure used. There are also other methods of representing rotations that are used in more complex situations, but this is the way we'll do it in this book. Figure 3.14 depicts a cube being rotated by 30 degrees around the Y-axis in its object space.

It is important to realize that the order of the rotations applied to the object matters a great deal. The convention we will use is the *roll-pitch-yaw* method, adopted from the aviation community. When we rotate the object, we roll it around its longitudinal (Z) axis. Then we pitch it around the lateral (X) axis. Finally, we yaw it around the vertical (Y) axis. Rotations on the object are applied in object space.

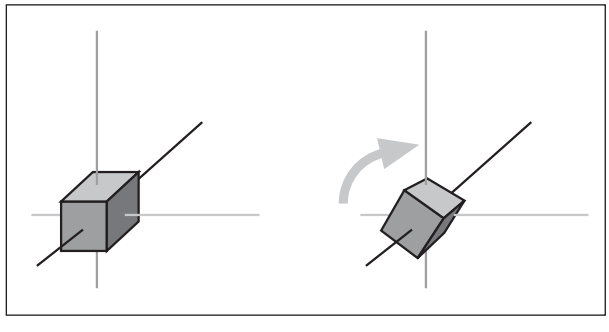


Figure 3.14 Rotation.

If we apply the rotation in a different order, we can end up with a very different orientation, despite having done the rotations using the same values.

Translation

Translation is the simplest of the transformations and the first that is applied to the object when transforming from object space to world space. Figure 3.15 shows a translation operation performed on an object. Note that the vertical axis is dark gray. As I said earlier, in this book, dark gray represents the Z-axis. Try to figure out what coordinate system we are using here. I'll tell you later in the chapter. To translate an object, we apply a vector to its position coordinates. Vectors can be specified in different ways, but the notation we will use is the same as the XYZ triplet, called a *vector triplet*. For Figure 3.15, the vector triplet is (3,9,7). This indicates that the object will be moved three units in the positive X direction, nine units in the positive Y direction, and seven units in the positive Z direction. Remember that this translation is applied in world space, so the X direction in this case would be eastward, and the Z direction would be down (toward the ground, so to speak). Neither the orientation nor the size of the object is changed.

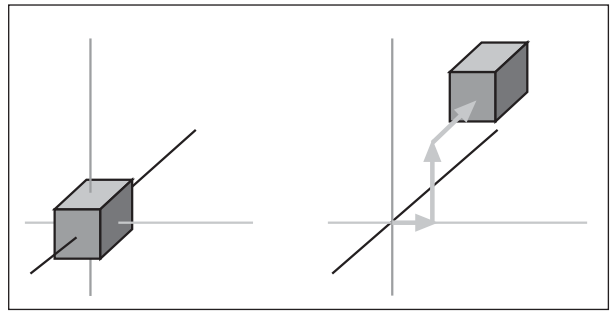


Figure 3.15 Translation.

Full Transformation

So now we roll all the operations together. We want to orient the cube a certain way, with a certain size, at a certain location. The transformations applied are scale (s)=1.6,1.6,1.6, followed by rotation (r)=0,30,0, and then finally translation (t)=3,9,7. Figure 3.16 shows the process.

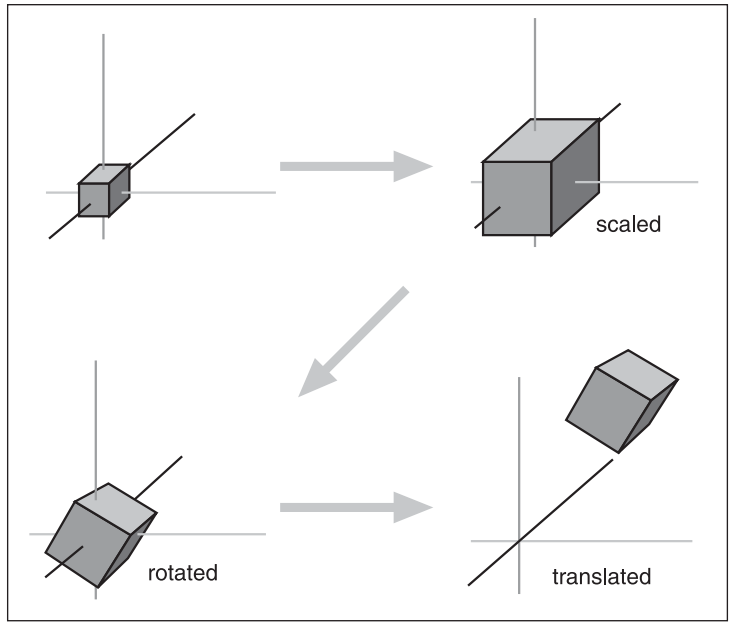


Figure 3.16 Fully transforming the cube.

note

The order that we use to apply the transformations is important. In the great majority of cases, the correct order is scaling, rotation, and then translation. The reason is that different things happen depending on the order.

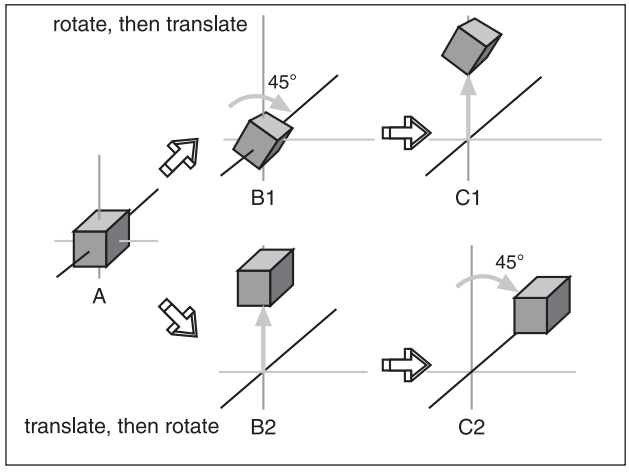


Figure 3.17 Faces on an irregularly shaped object.

You will recall that objects are created in object space, then moved into world space. The object's origin is placed at the world origin. When we rotate the object, we rotate it around the appropriate axes with the origin at (0,0,0), then translate it to its new position.

If you translate the object first, then rotate it (which is still going to take place around (0,0,0), the object will end up in an entirely different position as you can see in Figure 3.17.

Rendering

Rendering is the process of converting the 3D mathematical model of an object into an on-screen 2D image. When we render an object, our primary task is to calculate the appearance of the different faces of the object, convert those faces into a 2D form, and send the result to the video card, which will then take all the steps needed to display the object on your monitor.

We will take a look at several different techniques for rendering, those that are often used in video game engines or 3D video cards. There are other techniques, such as ray-casting, that aren't in wide use in computer games—with the odd exception, of course—that we won't be covering here.

In the previous sections our simple cube model had colored faces. In case you haven't noticed (but I'm sure you did notice), we haven't covered the issue of the faces, except briefly in passing.

A *face* is essentially a set of one or more contiguous co-planar adjacent triangles; that is, when taken as a whole, the triangles form a single flat surface. If you refer back to Figure 3.12, you will see that each face of the cube is made with two triangles. Of course, the faces are transparent in order to present the other parts of the cube.

Team LRN

Flat Shading

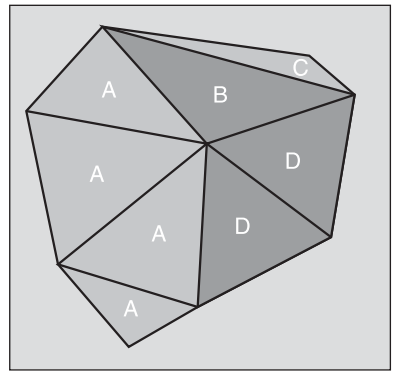


Figure 3.18 Faces on an irregularly shaped object.

Figure 3.18 provides an example of various face configurations on an irregularly shaped object. Each face is presented with a different color (which are visible as different shades). All triangles with the label A are part of the same face; the same applies to the D triangles. The triangles labeled B and C are each single-triangle faces.

When we want to display 3D objects, we usually use some technique to apply color to the faces. The simplest method is *flat shading*, as used in Figure 3.17. A color or shade is applied to a face, and a different color or shade is applied to adjacent faces so that the user can tell them apart. In this case, the shades were selected with the sole criterion being the need to distinguish one face from the other.

One particular variation of flat shading is called *Z-flat shading*. The basic idea is that the farther a face is from the viewer, the darker or lighter the face.

Lambert Shading

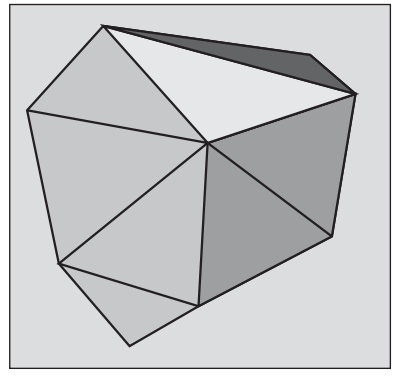


Figure 3.19 Lambert-shaded object.

Usually color and shading are applied in a manner that implies some sense of depth and lighted space. One face or collection of faces will be lighter in shade, implying that the direction they face has a light source. On the opposite side of the object, faces are shaded to imply that no light, or at least less light, reaches those faces. In between the light and dark faces, the faces are shaded with intermediate values. The result is a shaded object where the face shading provides information that imparts a sense of the object in a 3D world, enhancing the illusion. This is a form of flat shading known as *lambert shading* (see Figure 3.19).

Gouraud Shading

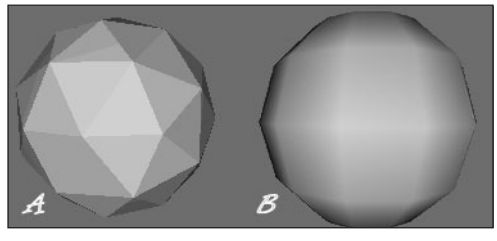


Figure 3.20 Flat-shaded (A) and gouraud-shaded (B) spheres.

A more useful way to color or shade an object is called *gouraud shading*. Take a look at Figure 3.20. The sphere on the left (A) is flat shaded, while the sphere on the right (B) is gouraud shaded. Gouraud shading smoothes the colors by averaging the *normals* (the vectors that indicate which way surfaces are facing) of the

vertices of a surface. The normals are used to modify the color value of all the pixels in a face. Each pixel's color value is then modified to account for the pixel's position within the face. Gouraud shading creates a much more natural appearance for the object, doesn't it? Gouraud shading is commonly used in both software and hardware rendering systems.

Phong Shading

Phong shading is a much more sophisticated—and computation-intensive—technique for rendering a 3D object. Like gouraud shading, it calculates color or shade values for each pixel. Unlike gouraud shading (which uses only the vertices' normals to calculate average pixel values), phong shading computes additional normals for each pixel between vertices and then calculates the new color values. Phong shading does a remarkably better job (see Figure 3.21), but at a substantial cost.

Phong shading requires a great deal of processing for even a simple scene, which is why you don't see phong shading used much in real-time 3D games where frame rate performance is important. However, there are games made where frame rate is not as big an issue, in which case you will often find phong shading used.

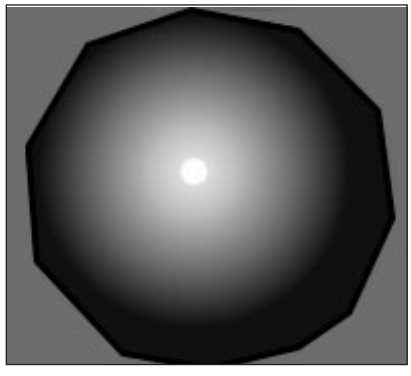


Figure 3.21 Phong-shaded sphere.

Fake Phong Shading

There is a rendering technique that looks almost as good as phong shading but can allow fast frame rates. It's called *fake phong shading*, or sometimes *fast phong shading*, or sometimes even *phong approximation rendering*. Whatever name it goes by, it is *not* phong rendering. It is useful, however, and does indeed give good performance.

Fake phong shading basically employs a bitmap, which is variously known as a *phong map*, a *highlight map*, a *shade map*, or a *light map*. I'm sure there are other names for it as well. In any event, the bitmap is nothing more than a generic template of how the faces should be illuminated (as shown in Figure 3.22).

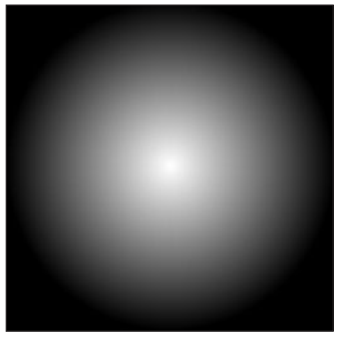


Figure 3.22 Example of a fake phong highlight map.

As you can tell by the nomenclature, there is no real consensus about fake phong shading. There are also several different algorithms used by different people. This diversity is no doubt the result of several people independently arriving at the same general concept at roughly the same time—all in search of better performance with high-quality shading.

Texture Mapping

Texture mapping is covered in more detail in Chapters 8 and 9. For the sake of completeness, I'll just say here that *texture mapping* an object is something like wallpapering a room. A 2D bitmap is "draped" over the object, to impart detail and texture upon the object, as shown in Figure 3.23.

Texture mapping is usually combined with one of the shading techniques covered in this chapter.

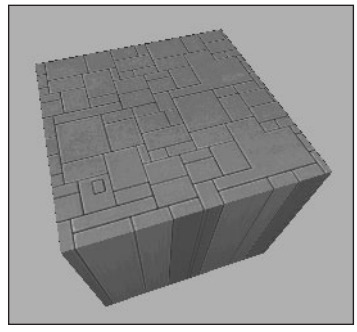


Figure 3.23 Texture-mapped and gouraud-shaded cube.

Shaders

When the word is used alone, *shaders* refers to *shader programs* that are sent to the video hardware by the software graphics engine. These programs tell the video card in great detail and procedure how to manipulate vertices or pixels, depending on the kind of shader used.

Traditionally, programmers have had limited control over what happens to vertices and pixels in hardware, but the introduction of shaders allowed them to take complete control.

Vertex shaders, being easier to implement, were first out of the starting blocks. The shader program on the video card manipulates vertex data values on a 3D plane via mathematical operations on an object's vertices. The operations affect color, texture coordinates, elevation-based fog density, point size, and spatial orientation.

Pixel shaders are the conceptual siblings of vertex shaders, but they operate on each discrete viewable pixel. Pixel shaders are small programs that tell the video card how to manipulate pixel values. They rely on data from vertex shaders (either the engine-specific custom shader or the default video card shader function) to provide at least triangle, light, and view normals.

Shaders are used in addition to other rendering operations, such as texture mapping.

Bump Mapping

Bump mapping is similar to texture mapping. Where texture maps *add* detail to a shape, bump maps *enhance* the shape detail. Each pixel of the bump map contains information that describes aspects of the physical shape of the object at the corresponding point, and we use a more expansive word to describe this—the *texel*. The name texel derives from *texture pixel*.

Bump mapping gives the illusion of the presence of bumps, holes, carving, scales, and other small surface irregularities. If you think of a brick wall, a texture map will provide the shape, color, and approximate roughness of the bricks. The bump map will supply a

detailed sense of the roughness of the brick, the mortar, and other details. Thus bump mapping enhances the close-in sense of the object, while texture mapping enhances the sense of the object from farther away.

Bump mapping is used in conjunction with most of the other rendering techniques.

Environment Mapping

Environment mapping is similar to texture mapping, except that it is used to represent effects where environmental features are reflected in the surfaces of an object. Things like chrome bumpers on cars, windows, and other shiny object surfaces are prime candidates for environment mapping.

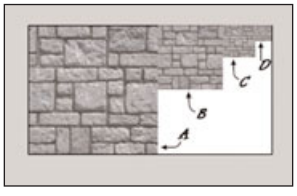


Figure 3.24 Mipmap textures for a stone surface.

Mipmapping

Mipmapping is a way of reducing the amount of computation needed to accurately texture-map an image onto a polygon. It's a rendering technique that tweaks the visual appearance of an object. It does this by using several different textures for the texture-mapping operations on an object. At least two, but usually four, textures of progressively lower resolution are assigned to any given surface, as shown in Figure 3.24. The video card or graphics engine extracts pixels from each texture, depending on the distance and orientation of the surface compared to the view screen.

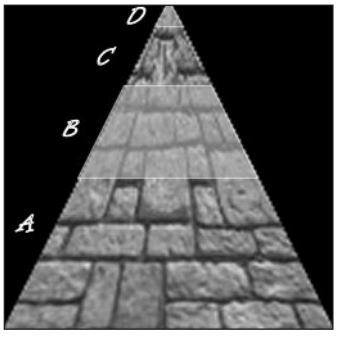


Figure 3.25 Receding mipmap textures on a stone surface.

In the case of a flat surface that recedes away from the viewer into the distance, for pixels on the nearer parts of the surface, pixels from the high-resolution texture are used (see Figure 3.25). For the pixels in the middle distances, pixels from the medium-resolution textures are used. Finally, for the faraway parts of the surface, pixels from the low-resolution texture are used.

tip

Anti-aliasing is a software technique used in graphics display systems to make curved and diagonal lines appear to be continuous and smooth. On computer monitors the pixels themselves aren't curved, but collectively they combine together to represent curves. Using pixels within polygon shapes to simulate curves causes the edges of objects to appear jagged. Anti-aliasing, the technique for smoothing out these jaggies, or aliasing, usually takes the form of inserting intermediate-colored pixels along the edges of the curve. The funny thing is, with textual displays this has the paradoxical effect of making text blurrier yet more readable. Go figure!

Scene Graphs

In addition to knowing how to construct and render 3D objects, 3D engines need to know how the objects are laid out in the virtual world and how to keep track of changes in status of the models, their orientation, and other dynamic information. This is done using a mechanism called a *scene graph*, a specialized form of a *directed graph*. The scene graph maintains information about all entities in the virtual world in structures called *nodes*. The 3D engine traverses this graph, examining each node one at a time to determine how to render each entity in the world. Figure 3.26 shows a simple seaside scene with its scene graph. The nodes marked by ovals are *group nodes*, which contain information about themselves and point to other nodes. The nodes that use rectangles are *leaf nodes*. These nodes contain only information about themselves.

Note that in the seaside scene graph, not all of the nodes contain all of the information that the other nodes have about themselves.

Many of the entities in a scene don't even need to be rendered. In a scene graph, a node can be anything. The most common entity types are 3D shapes, sounds, lights (or lighting information), fog and other environmental effects, viewpoints, and event triggers.

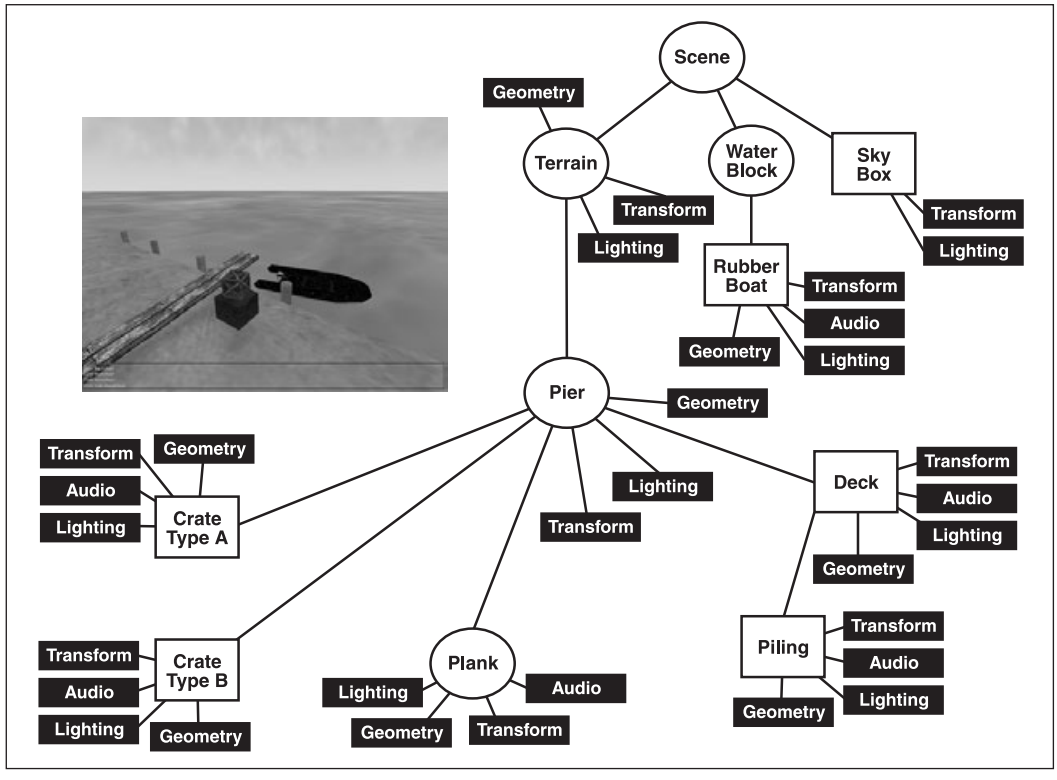


Figure 3.26 Simple scene graph.

When it comes time to render the scene, the Torque Engine will "walk" through the nodes in the tree of the scene graph, applying whatever functions to the node that are specified. It then uses the node pointers to move on to the next node to be rendered.

3D Audio

Audio and sound effects are used to heighten the sense of realism in a game. There are times when the illusion is greatly enhanced by using position information when generating the sound effects. A straightforward example would be the sound generated by a nearby gunshot. By calculating the amplitude—based on how far away the shot occurred—and the direction, the game software can present the sound to a computer's speakers in a way that gives the player a strong sense of where the shot occurred. This effect is even better if the player is wearing audio headphones. The player then has a good sense of the nature of any nearby threat and can deal with it accordingly—usually by massive application of return fire.

The source location of a game sound is tracked and managed in the same way as any other 3D entity via the scene graph.

Once the game engine has decided that the sound has been triggered, it then converts the location and distance information of the sound into a stereo "image" of the sound, with appropriate volumes and balance for either the right or left stereo channel. The methods used to perform these calculations are much the same as those used for 3D object rendering.

Audio has an additional set of complications—things like fade and drop-off or cutoff.

3D Programming

With the Torque Engine, most of the really grubby low-level programming is done for you. Instead of writing program code to construct a 3D object, you use a modeling tool (which we cover in later chapters) to create your object and a few lines of script code to insert the object in a scene. You don't even need to worry about where in the scene graph the object should be inserted—Torque handles that as well, through the use of information contained in the datablocks that you define for objects.

Even functions like moving objects around in the world are handled for us by Torque, simply by defining the object to be of a certain class and then inserting the object appropriately.

The kinds of objects we will normally be using are called *shapes*. In general, shapes in Torque are considered to be dynamic objects that can move or otherwise be manipulated by the engine at run time.

There are many shape classes; some are fairly specific, like vehicles, players, weapons, and projectiles. Some are more general-purpose classes, like items and static shapes. Many of

the classes know how their objects should respond to game stimuli and are able to respond in the game with motion or some other behavior inherent to the object's class definition.

Usually, you will let the game engine worry about the low-level mechanics of moving your 3D objects around the game world. However, there will probably be times while creating a game that you are going to want to cause objects to move in some nonstandard way—some method not defined by the class definition of the object. With Torque, this is easy to do!

Programmed Translation

When an object in 3D world space moves, it is *translating* its position, in a manner similar to that shown earlier in the discussion about transformations.

You don't, however, absolutely need to use the built-in classes to manipulate shapes in your game world. For example, you can write code to load in an *Interior* (a class of objects used for structures like buildings) or an *Item* (a class of objects used for smaller mobile and static items in a game world, like signs, boxes, and powerups). You can then move that object around the world any way you like.

You can also write code to monitor the location of dynamic shapes that are moving around in the world, detect when they reach a certain location, and then arbitrarily move, or *teleport*, those objects to some other location.

Simple Direct Movement

What we are going to do is select an object in a 3D scene in Torque and then move it from one location to another using some script instructions entered directly into the game console. The first step is to identify the object.

1. In the 3DGPAi1 folder locate the Run Chapter 3 shortcut and double-click it to launch the demo.
2. Click Start.
3. Using the mouse, turn your player-character to the left or right a bit, if necessary, until you have a good view of the pyramid.
4. Press F11. Torque's built-in World Editor will appear. As you move your cursor over the scene, you'll notice it change to a hand icon.
5. Click the hand on the pyramid to select it.
6. Move the cursor over to the right side, and click once on the plus sign to the left of the words "MissionGroup - SimGroup". You will see the list expand, and one of the entries, of the type InteriorInstance, will be highlighted. Take note of the number to the left, which is the object's instance ID. See Figure 3.27 for help, if necessary. The ID I get from the figure is 1359; your result should be the same.

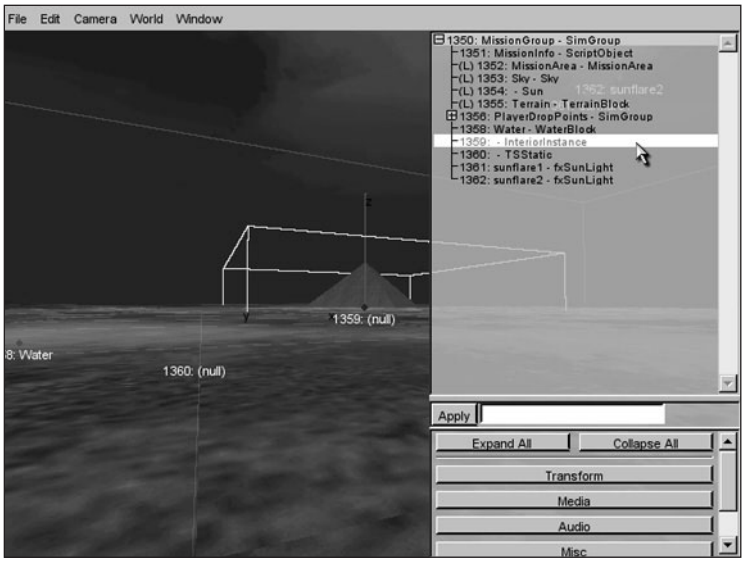


Figure 3.27 Finding the pyramid object's instance ID.

7. Press the Tilde ("~") key, and the console will pop open. The console interface allows us to directly type in program code and get immediate results.
8. In the console window, type **echo(1353.getTransform());** and then press the Enter key. Don't forget to include the semicolon at the end of the line before you press the Enter key.

   You should get a result like 49.2144 -66.1692 0.4 0 0 -1 9.74027, which is the transform of the pyramid. The first three numbers are the XYZ coordinates of the geometric center of the pyramid. The next three are the axis normals, which in this case indicates that the Z-axis is pointing straight up. The final value indicates how much rotation is applied around the rotation axes. We'll look at rotation in more detail a little later. Here, the rotation amount (in degrees) is applied to only the Z-axis.
9. In the console window, type **1353.setTransform("0 0 190 0 0 1 0");** and then press the Enter key.
10. Press the Escape key to remove the console window, and take a look. You will notice that the pyramid has moved.
11. Take the next several minutes to experiment with different transforms. Try rotating the pyramid around different axes or several axes at the same time.
12. When you are done, press the Tilde key to exit the console window, press Escape to exit the World Editor, and then press Escape one more time to exit the game.

tip

In the little exercise in the "Simple Direct Movement" section, you saw a command that looked like this: `echo(1353.getTransform() );`. The number 1353 is an object ID, and the `getTransform()` part is what is called a *method* of that object. A method is a function that belongs to a specific object *class*. We'll cover these topics in more detail in a later chapter.

Programmed Movement

Now we are going to explore how we can move things in the 3D world using program code. We are going to use the Item class to create an object based on a model of a stylized heart, insert the object in the game world, and then start it slowly moving across the terrain—all using Torque Script.

Something to know about the Item class is that Torque defines it as being affected by gravity. So if we insert the object into the scene at some distance above the ground level of the terrain, the object will actually fall to the ground—a little more slowly than it would in the real world, but what the hey! It's a game, after all. Anyway, this also means that we have to specify a mass and a friction drag value in order to prevent the item from sliding down hills if it lands on a slope.

Okay, now—so on to the program. Type the following code module into a file and save the file as 3DGPAi1\CH3\moveshape.cs.

```
// ============================================================================
//  moveshape.cs
//
//  This module contains the definition of a test shape, which uses
//  a model of a stylized heart. It also contains functions for placing
//  the test shape in the game world and moving the shape.
// ============================================================================

datablock ItemData(TestShape)
// ----------------------------------------------------
//     Definition of the shape object
// ----------------------------------------------------
{
   // Basic Item properties
   shapeFile = "~/data/shapes/items/heart.dts";
   mass = 1;       //we give the shape mass and
   friction = 1;  // friction to stop the item from sliding
                   // down hills
};

function InsertTestShape()
// ----------------------------------------------------
//     Instantiates the test shape, then inserts it
//     into the game world roughly in front of
//     the player's default spawn location.
// ----------------------------------------------------
{
```

```
   // An example function which creates a new TestShape object
   %shape = new Item() {
      datablock = TestShape;
      rotation = "0 0 1 0"; // initialize the values
                            // to something meaningful
   };
   MissionCleanup.add(%shape);

   // Player setup
   %shape.setTransform("-90 -2 20 0 0 1 0");
   echo("Inserting Shape " @ %shape);
   return %shape;
}

function MoveShape(%shape, %dist)
// ------------------------------------------------
//     moves the %shape by %dist amount
// ------------------------------------------------
{
   %xfrm = %shape.getTransform();
   %lx = getword(%xfrm,0); // get the current transform values
   %ly = getword(%xfrm,1);
   %lz = getword(%xfrm,2);
   %lx += %dist;           // adjust the x axis position
   %shape.setTransform(%lx SPC %ly SPC %lz SPC "0 0 1 0");
}

function DoMoveTest()
// ------------------------------------------------
//     a function to tie together the instantiation
//     and the movement in one easy to type function
//     call.
// ------------------------------------------------
{
   %ms = InsertTestShape();
   MoveShape(%ms,15);
}
```

In this module there are three functions and a *datablock* definition. A datablock is a construct that is used to organize properties for objects together in a way that is important for the server. We will cover datablocks in more detail in a later chapter. The datablock begins with the line `datablock ItemData(TestShape)`—it specifies static properties of the Item class that can't be changed while the game is running. The most important part of the preceding

datablock is the `shapeFile` property, which tells Torque where to find the model that will be used to represent the object. The mass and friction values, as mentioned previously, prevent the item from sliding down hills because of the pernicious tug of gravity.

The function `InsertTestShape()` creates a new instance of `TestShape` with the call to `new Item()`. It specifies the `TestShape` datablock, defined earlier, and then sets the object's rotation to some sensible values.

Next, `MissionCleanup.add(%shape);` adds the shape instance to a special mission-related group maintained by Torque. When the mission ends, objects assigned to this group are deleted from memory (cleaned up) before a new mission is started.

After that, the program sets the initial location of the object by setting the transform.

Next, the `echo` statement prints the shape's handle to the console.

Finally, the shape's handle (ID number) is returned from the function. This allows us to save the handle in a variable when we call this function, so that we can refer to this same item instance at a later time.

The function `MoveShape` accepts a shape handle and a distance as arguments, and uses these to move whatever shape the handle indicates.

First, it gets the current position of the shape using the `%shape.getTransform()` method of the Item class.

Next, the program employs the `getword()` function to extract the parts of the transform string that are of interest and store them in local variables. We do this because, for this particular program, we want to move the shape in the X-axis. Therefore, we strip out all three axes and increment the X value by the distance that the object should move. Then we prepend all three axis values to a dummy rotation and set the item's transform to be this new string value. This last bit is done with the `%shape.setTransform()` statement.

The `DoMoveTest()` function is like a wrapper folded around the other functions. When we call this function, first it inserts the new instance of the shape object using the `InsertTestShape()` function and saves the handle to the new object in the variable `%ms`. It then calls the `MoveShape()` function, specifying which shape to move by passing in the handle to the shape as the first argument and also indicating the distance with the second argument.

To use the program, follow these steps:

1. Make sure you've saved the file as 3DGPAi1\CH3\moveshape.cs.
2. Run the Chapter 3 demo using the shortcut in the 3DGPAi1 folder.
3. Press the Start button when the CH3 demo screen comes up.
4. Make sure you don't move your player-character after it spawns into the game world.

5. Bring up the console window by pressing the Tilde key.
6. Type in the following, and press Enter after the semicolon:

   ```
   exec("CH3/moveshape.cs");
   ```

   You should get a response in the console window similar to this:

   ```
   Compiling CH3/moveshape.cs...
   Loading compiled script CH3/moveshape.cs.
   ```

   This means that the Torque Engine has compiled your program and then loaded it into memory. The datablock definition and the three functions are in memory, waiting with barely suppressed anticipation for your next instruction.

tip

About those slashes… You've probably noticed that when you see the file names and paths written out, the back slash ("\") is used, and when you type in those same paths in the console window, the forward slash ("/") is used. This is not a mistake. Torque is a cross-platform program that is available for Macintosh and Linux as well as Windows. It's only on Windows-based systems that back slashes are used—everyone else uses forward slashes.

Therefore, the back slashes for Windows-based paths are the exception here. Just thought I'd clear that up!

7. Type the following into the console window:

   ```
   $tt = InsertTestShape();
   ```

   You should see the following response:

   ```
   Inserting Shape 1388
   ```

   The number you get may be different—that's not an issue. But it will probably be the same. Take note of the number.

   You also may see a warning about not locating a texture—that's of no importance here either.
8. Close the console window. You should see a heart on the ground in front of your player.
9. Type the following into the console:

   ```
   echo($tt);
   ```

   Torque will respond by printing the contents of the variable `$tt` to the console window. It should be the same number that you got as a response after using the `InsertTestShape` instruction above.
10. Type the following into the console:

    ```
    MoveShape(%tt,50);
    ```

11. Press the Tilde key to close the console window. You should see the heart move away from you to the left.

    You should be familiar with opening and closing the console window by now, so I won't bother explaining that part in the instruction sequences anymore.

12. Now, type this into the console, and close the console quickly afterward:

    ```
    DoMoveTest();
    ```

    What you should see now is the heart dropping from the air to the ground; it then moves away just like the first heart, except not as far this time.

The reason why the heart drops from the air is because the object's initial location was set in the `InsertTestShape()` function to be `-90 -2 20`, where the Z value is set to 20 units up. As mentioned earlier, Torque will automatically make objects of the Item class fall under gravity until they hit something that stops the fall. If you don't close the console window quickly enough, you won't see it fall.

Go ahead and experiment with the program. Try moving the item through several axes at once, or try changing the distance.

Programmed Rotation

As you've probably figured out already, we can rotate an object programmatically (or directly, for that matter) using the same `setTransform()` method that we used to translate an object.

Type the following program and save it as 3DGPAi1\CH3\turnshape.cs.

```
// ============================================================================
//  turnshape.cs
//
//  This module contains the definition of a test shape.
//  It contains functions for placing
//  the test shape in the game world and rotating the shape
// ============================================================================

datablock ItemData(TestShape)
// -----------------------------------------------------
//     Definition of the shape object
// -----------------------------------------------------
{
   // Basic Item properties
   shapeFile = "~/data/shapes/items/heart.dts";
   mass = 1;       //we give the shape mass and
   friction = 1;  // friction to stop the item from sliding
```

```
                    // down hills
};

function InsertTestShape()
// -------------------------------------------------
//     Instantiates the test shape, then inserts it
//     into the game world roughly in front of
//     the player's default spawn location.
// -------------------------------------------------
{
   // An example function which creates a new TestShape object
   %shape = new Item() {
      datablock = TestShape;
      rotation = "0 0 1 0"; // initialize the values
                            // to something meaningful
   };
   MissionCleanup.add(%shape);

   // Player setup
   %shape.setTransform("-90 -2 20 0 0 1 0");
   echo("Inserting Shape " @ %shape);
   return %shape;
}

function TurnShape(%shape, %angle)
// -------------------------------------------------
//     turns the %shape by %angle amount.
// -------------------------------------------------
{
   %xfrm = %shape.getTransform();
   %lx = getword(%xfrm,0); // first, get the current transform values
   %ly = getword(%xfrm,1);
   %lz = getword(%xfrm,2);
   %rx = getword(%xfrm,3);
   %ry = getword(%xfrm,4);
   %rz = getword(%xfrm,5);
   %angle += 1.0;
   %rd = %angle;            // Set the rotation angle
   %shape.setTransform(%lx SPC %ly SPC %lz SPC %rx SPC %ry SPC %rz SPC %rd);
}

function DoTurnTest()
```

```
// ------------------------------------------------
//     a function to tie together the instantiation
//     and the movement in one easy to type function
//     call.
// ------------------------------------------------
{
   %ts = InsertTestShape();
   TurnShape(%ts,30);
}
```

The program is quite similar to the moveshape.cs program that you were just working with. You can load and run the program in exactly the same way, except that you want to use `DoTurnTest()` instead of `DoMoveTest()` and `TurnShape()` instead of `MoveShape()`.

Things of interest to explore are the variables `%rx`, `%ry`, `%rz`, and `%rd` in the `TurnShape()` function. Try making changes to each of these and observing the effects your changes have on the item.

Programmed Scaling

We can also quite easily change the scale of an object using program code.

Type the following program and save it as 3DGPAi1\CH3\sizeshape.cs.

```
// ============================================================================
//  Sizeshape.cs
//
//  This module contains the definition of a test shape, which uses
//  a model of a stylized heart. It also contains functions for placing
//  the test shape in the game world and then sizing the shape.
// ============================================================================

datablock ItemData(TestShape)
// ------------------------------------------------
//     Definition of the shape object
// ------------------------------------------------
{
   // Basic Item properties
   shapeFile = "~/data/shapes/items/heart.dts";
   mass = 1;       //we give the shape mass and
   friction = 1;  // friction to stop the item from sliding
                   // down hills
};

function InsertTestShape()
```

```
// ----------------------------------------------------
//     Instantiates the test shape, then inserts it
//     into the game world roughly in front of
//     the player's default spawn location.
// ----------------------------------------------------
{
   // An example function which creates a new TestShape object
   %shape = new Item() {
      datablock = TestShape;
      rotation = "0 0 1 0"; // initialize the values
                            // to something meaningful
   };
   MissionCleanup.add(%shape);

   // Player setup
   %shape.setTransform("-90 -2 20 0 0 1 0");
   echo("Inserting Shape " @ %shape);
   return %shape;
}

function SizeShape(%shape, %scale)
// ----------------------------------------------------
//     moves the %shape by %scale amount
// ----------------------------------------------------
{
   %shape.setScale(%scale SPC %scale SPC %scale);
}

function DoSizeTest()
// ----------------------------------------------------
//     a function to tie together the instantiation
//     and the movement in one easy to type function
//     call.
// ----------------------------------------------------
{
   %ms = InsertTestShape();
   SizeShape(%ms,5);
}
```

The program is obviously similar to the moveshape.cs and turnshape.cs programs. You can load and run the program in exactly the same way, except that you want to use `DoSizeTest()` instead of `DoMoveTest()` and `SizeShape()` instead of `MoveShape()`.

You'll note that we don't call the object's `%shape.getScale()` function (there is one), because in this case, we don't need to. Also notice that the three arguments to our call to `%shape.setScale()` all use the same value. This is to make sure the object scales equally in all dimensions. Try making changes to each of these and observing the effects your changes have on the item.

Another exercise would be to modify the `SizeShape` function to accept a different parameter for each dimension (X, Y, or Z) so that you can change all three to different scales at the same time.

Programmed Animation

You can animate objects by stringing together a bunch of translation, rotation, and scale operations in a continuous loop. Like the transformations, most of the animation in Torque can be left up to an object's class methods to perform. However, you can create your own ad hoc animations quite easily by using the `schedule()` function.

Type the following program and save it as 3DGPAi1\CH3\animshape.cs.

```
// ============================================================================
//  animshape.cs
//
//  This module contains the definition of a test shape, which uses
//  a model of a stylized heart. It also contains functions for placing
//  the test shape in the game world and then animating the shape using
//  a recurring scheduled function call.
// ============================================================================

datablock ItemData(TestShape)
// -----------------------------------------------------
//     Definition of the shape object
// -----------------------------------------------------
{
   // Basic Item properties
   shapeFile = "~/data/shapes/items/heart.dts";
   mass = 1;       //we give the shape mass and
   friction = 1;  // friction to stop the item from sliding
                   // down hills
};

function InsertTestShape()
// -----------------------------------------------------
//     Instantiates the test shape, then inserts it
//     into the game world roughly in front of
```

```
//     the player's default spawn location.
// -----------------------------------------------------
{
   // An example function which creates a new TestShape object
   %shape = new Item() {
      datablock = TestShape;
      rotation = "0 0 1 0"; // initialize the values
                            // to something meaningful
   };
   MissionCleanup.add(%shape);

   // Player setup
   %shape.setTransform("-90 -2 20 0 0 1 0");
   echo("Inserting Shape " @ %shape);
   return %shape;
}

function AnimShape(%shape, %dist, %angle, %scale)
// -----------------------------------------------------
//     moves the %shape by %dist amount, and then
//     schedules itself to be called again in 1/5
//     of a second.
// -----------------------------------------------------
{

   %xfrm = %shape.getTransform();
   %lx = getword(%xfrm,0); // first, get the current transform values
   %ly = getword(%xfrm,1);
   %lz = getword(%xfrm,2);
   %rx = getword(%xfrm,3);
   %ry = getword(%xfrm,4);
   %rz = getword(%xfrm,5);
   %lx += %dist;            // set the new x position
   %angle += 1.0;
   %rd = %angle;            // Set the rotation angle

   if ($grow)              // if the shape is growing larger
   {
```

```
    if (%scale < 5.0)      // and hasn't gotten too big
      %scale += 0.3;       // make it bigger
    else
      $grow = false;       // if it's too big, don't let it grow more
  }
  else                    // if it's shrinking
  {
    if (%scale > 3.0)  // and isn't too small
      %scale -= 0.3;   // then make it smaller
    else
      $grow = true;   // if it's too small, don't let it grow smaller
  }

  %shape.setScale(%scale SPC %scale SPC %scale);
  %shape.setTransform(%lx SPC %ly SPC %lz SPC %rx SPC %ry SPC %rz SPC %rd);
  schedule(200,0,AnimShape, %shape, %dist, %angle, %scale);

}

function DoAnimTest()
// ------------------------------------------------
//     a function to tie together the instantiation
//     and the movement in one easy to type function
//     call.
// ------------------------------------------------
{
  %as = InsertTestShape();
  $grow = true;
  AnimShape(%as,0.2, -1, -2);
}
```

This module is almost identical to the `MoveShape()` module we worked with earlier.

The function `AnimShape` accepts a shape handle in `%shape`, a distance step as `%dist`, an angle value as `%angle`, and a scaling value as `%scale` and uses these to transform the shape indicated by the `%shape` handle.

First, it obtains the current position of the shape using the `%shape.getTransform()` method of the Item class.

As with the earlier `MoveShape()` function, the `AnimShape()` function fetches the transform of the shape and updates one of the axis values.

Then it updates the rotation value stored `%rd`.

Then it adjusts the scale value by determining if the shape is growing or shrinking. Depending on which way the size is changing, the scale is incremented, unless the scale exceeds the too large or too small limits. When a limit is exceeded, the change direction is reversed.

Next, the scale of the shape is changed to the new values using the `%shape.setScale()` method for the shape.

Finally, the function sets the item's transform to be the new transform values within the `%shape.setTransform()` statement.

The `DoAnimTest()` function first inserts the new instance of the shape object using the `InsertTestShape()` function and saves the handle to the new object in the variable `%as`. It then calls the `AnimShape()` function, specifying which shape to animate by passing in the handle to the shape as the first argument and also indicating the discrete movement step distance, the discrete rotation angle, and the discrete size change value with the second, third, and fourth arguments.

To use the program, follow these steps:

1. Make sure you've saved the file as 3DGPAi1\CH3\animshape.cs.
2. Run the Chapter 3 demo using the shortcut in the 3DGPAi1 folder.
3. Press the Start button when the demo screen comes up.
4. Make sure you don't move your player-character after it spawns into the game world.
5. Bring up the console window.
6. Type in the following, and press Enter after the semicolon:

   ```
   exec("CH3/animshape.cs");
   ```

   You should get a response in the console window similar to this:

   ```
   Compiling CH3/animshape.cs...
   Loading compiled script CH3/animshape.cs.
   ```

   This means that the Torque Engine has compiled your program and then loaded it into memory. The datablock definition and the three functions are in memory, waiting to be used.
7. Now, type the following into the console, and close the console quickly afterward:

   ```
   DoAnimTest();
   ```

   What you should see now is the heart dropping from the air to the ground; it then begins moving away from you toward the right. Go chase after it if you like, to get a sense of how fast it is moving.

Go ahead and experiment with the program. Try moving the item through several axes at once, or try changing the distance.

3D Audio

Environmental sounds with a 3D component contribute greatly to the immersive aspect of a game by providing positional cues that mimic the way sounds happen in real life.

We can control 3D audio in the scene in much the same way we do 3D visual objects.

Type the following program and save it as 3DGPAi1\CH3\animaudio.cs.

```
// ============================================================================
//  animaudio.cs
//
//  This module contains the definition of an audio emitter, which uses
//  a synthetic water drop sound. It also contains functions for placing
//  the test emitter in the game world and moving the emitter.
// ============================================================================

datablock AudioProfile(TestSound)
// ----------------------------------------------------
//     Definition of the audio profile
// ----------------------------------------------------
{
   filename = "~/data/sound/testing.wav"; // wave file to use for the sound
   description = "AudioDefaultLooping3d"; // monophonic sound that repeats
        preload = false;  // Engine will only load sound if it encounters it
                          // in the mission
};

function InsertTestEmitter()
// ----------------------------------------------------
//     Instantiates the test sound, then inserts it
//     into the game world to the right and offset somewhat
//     from the player's default spawn location.
// ----------------------------------------------------
{
   // An example function which creates a new TestSound object
   %emtr = new AudioEmitter() {
      position = "0 0 0";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      profile = "TestSound"; // Use the profile in the datablock above
```

```
      useProfileDescription = "1";
      type = "2";
      volume = "1";
      outsideAmbient = "1";
      referenceDistance = "1";
      maxDistance = "100";
      isLooping = "1";
      is3D = "1";
      loopCount = "-1";
      minLoopGap = "0";
      maxLoopGap = "0";
      coneInsideAngle = "360";
      coneOutsideAngle = "360";
      coneOutsideVolume = "1";
      coneVector = "0 0 1";
      minDistance = "20.0";
   };
   MissionCleanup.add(%emtr);

   // Player setup-
   %emtr.setTransform("-200 -30 12 0 0 1 0"); // starting location
   echo("Inserting Audio Emitter " @ %emtr);
   return %emtr;
}

function AnimSound(%snd, %dist)
// ------------------------------------------------
//     moves the %snd by %dist amount each time
// ------------------------------------------------
{
   %xfrm = %snd.getTransform();
   %lx = getword(%xfrm,0); // first, get the current transform values
   %ly = getword(%xfrm,1);
   %lz = getword(%xfrm,2);
   %rx = getword(%xfrm,3);
   %ry = getword(%xfrm,4);
   %rz = getword(%xfrm,5);
   %lx += %dist;            // set the new x position
   %snd.setTransform(%lx SPC %ly SPC %lz SPC %rx SPC %ry SPC %rz SPC %rd);
   schedule(200,0,AnimSound, %snd, %dist);

}
```

```
function DoMoveTest()
// ------------------------------------------------
//     a function to tie together the instantiation
//     and the movement in one easy to type function
//     call.
// ------------------------------------------------
{
   %ms = InsertTestEmitter();
   AnimSound(%ms,1);
}
DoMoveTest();   // by putting this here, we cause the test to start
                // as soon as this module has been loaded into memory
```

In this program, we also have a datablock, but you'll notice that it is different this time. This datablock defines an *audio profile*. It contains the name of the wave file that contains the sound to be played, a descriptor that tells Torque how to treat the sound, and a flag to indicate whether the engine should automatically load the sound or wait until it encounters a need for the sound. In this case, the engine will wait until it knows it needs the file.

The `InsertTestEmitter` function is structured the same as the earlier `InsertTestShape` function, but this time it creates the object with a call to `new AudioEmitter`, and there are quite a few properties to be set. These properties will be explained in much greater detail in Chapters 19 and 20.

Another difference to note is the last line, which is a call to `DoMoveTest`. This allows us to load and run the program in one go, using the exec call. After the Torque Engine compiles the program, it loads it into memory and runs through the code. In our earlier program, like the `AnimShape` module, Torque would encounter the datablock and function definitions. Because they are definitions, they aren't executed, just loaded into memory. The last line, however, is not a definition. It is a statement that calls a function. So when Torque encounters it, Torque looks to see if it has the function resident in memory, and if so, it executes the function according to the syntax of the statement.

To use the program, follow these steps:

1. Make sure you've saved the file as 3DGPAi1\CH3\ animaudio.cs.
2. Run the Chapter 3 demo using the shortcut in the 3DGPAi1 folder.
3. Press the Start button when the demo screen comes up.
4. Make sure you don't move your player-character after it spawns into the game world.
5. Bring up the console window.
6. Type in the following, and press Enter after the semicolon:

   ```
   exec("CH3/animaudio.cs");
   ```

You should get a response in the console window similar to this:

```
Compiling CH3/animaudio.cs...
Loading compiled script CH3/animaudio.cs.
```

You should also begin to hear the dripping sound off to the right-hand side. If you wait without moving your player in any way, not even using the mouse to turn his head, you will notice the sound slowly approach you from the left, pass over to the right in front of you, and then pass off into the distance to the left. Pretty neat, huh?

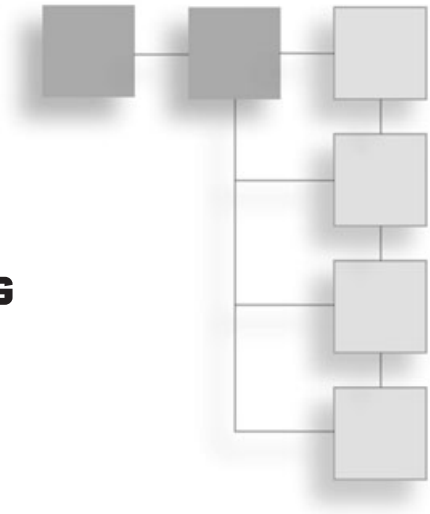
Moving Right Along

So, we've now seen how 3D objects are constructed from vertices and faces, or polygons. We explored how they fit into that virtual game world using transformations and that the transformations are applied in particular order—scaling, rotation, and then finally translation. We also saw how different rendering techniques can be used to enhance the appearance of 3D models.

Then we learned practical ways to apply those concepts using program code written using Torque Script and tested with the Torque Game Engine.

In the next chapter, we will dive deeper into learning how to use Torque Script.

CHAPTER 4



Game Programming

In the preceding two chapters you were introduced to a few new concepts: programming, 3D graphics, manipulating 3D objects, and stuff like that. Most of it was fairly broad, in order to give you a good grasp of what you can do to make your game.

The next bunch of chapters get down and dirty, so to speak. We're going to muck around with our own hands examining things, creating things, and making things happen.

In this chapter we're going to hammer at the Torque Script for a while, writing actual code that will be used to develop our game. We'll examine in detail how the code works in order to gain a thorough understanding of how Torque works. The game we are going to create has the rather unoriginal name of *Emaga*, which is just *agame* spelled backward. The Chapter 4 version will be called *Emaga4*. Of course, you may—and probably should—substitute whatever name you wish!

Torque Script

As I've said before, Torque Script is much like C/C++, but there are a few differences. Torque Script is typeless—with a specific exception regarding the difference between numbers and strings—and you don't need to pre-allocate storage space with variable declarations.

All aspects of a game can be controlled through the use of Torque Script, from game rules and nonplayer character behavior to player scoring and vehicle simulation. A script comprises *statements*, *function declarations*, and *package declarations*.

Most of the syntax in *Torque Game Engine* (TGE) Script language is similar to C/C++ language, with a high correlation of keywords (see Table A.3 in Appendix A) between the two. Although, as is often the case in scripting languages, there is no type enforcement on the

Team LRN

variables, and you don't declare variables before using them. If you read a variable before writing it, it will be an empty string or zero, depending on whether you are using it in a string context or a numeric context.

The engine has rules for how it converts between the script representation of values and its own internal representation. Most of the time the correct script format for a value is obvious; numbers are numbers (also called *numerics*), strings are strings, the tokens `true` and `false` can be used for ease-of-code-reading to represent 1 and 0, respectively. More complicated data types will be contained within strings; the functions that use the strings need to be aware of how to interpret the data in the strings.

Strings

String constants are enclosed in single quotes or double quotes. A single-quoted string specifies a *tagged* string—a special kind of string used for any string constant that needs to be transmitted across a connection. The full string is sent once, the first time. And then whenever the string needs to be sent again, only the short tag identifying that string is sent. This dramatically reduces bandwidth consumption by the game.

A double-quoted (or *standard*) string is not tagged; therefore, whenever the string is used, storage space for all of the characters contained in the string must be allocated for whatever operation the string is being used for. In the case of sending a standard string across connections, all of the characters in the string are transmitted, every single time the string is sent. Chat messages are sent as standard strings, and because they change each time they are sent, creating tag ID numbers for chat messages would be pretty useless.

Strings can contain formatting codes, as described in Table 4.1.

Table 4.1 Torque Script String Formatting Codes

Code	Description
`\r`	Embeds a carriage return character.
`\n`	Embeds a new line character.
`\t`	Embeds a tab character.
`\xhh`	Embeds an ASCII character specified by the hex number (*hh*) that follows the *x*.
`\c`	Embeds a color code for strings that will be displayed on-screen.
`\cr`	Resets the display color to the default.
`\cp`	Pushes the current display color onto a stack.
`\co`	Pops the current display color off the stack.
`\cn`	Uses *n* as an index into the color table defined by `GUIControlProfile.fontColors`.

Objects

Objects are instances of object classes, which are a collection of properties and methods that together define a specific set of behaviors and characteristics. A Torque object is an *instantiation* of an object class. After creation, a Torque object has a unique numeric identifier called its *handle*. When two handle variables have the same numeric value, they refer to the same object. An *instance* of an object can be thought of as being somewhat like a *copy* of an object.

When an object exists in a multiplayer game with a server and multiple clients, the server and each client allocate their own handle for the object's storage in memory. Note that datablocks (a special kind of object) are treated differently—more about this a little later.

note

Methods are functions that are accessible through objects. Different object classes may have some methods that are common between them, and they may have some methods that are unique to themselves. In fact, methods may have the same name, but work differently, when you move from one object class to another.

Properties are variables that belong to specific objects and, like methods, are accessed through objects.

Creating an Object

When creating a new instance of an object, you can initialize the object's fields in the `new` statement code block, as shown here:

```
%handle = new InteriorInstance()
{
   position = "0 0 0";
   rotation = "0 0 0";
   interiorFile = %name;
};
```

The handle of the newly created `InteriorInstance` object is inserted into the variable `%handle` when the object is created. Of course, you could use any valid and unused variable you want, like `%obj`, `%disTing`, or whatever. Note in the preceding example that `%handle` is a local variable, so it is only in scope—or valid—within the function where it is used. Once the memory is allocated for the new object instance, the engine then initializes the object's properties as directed by the program statements embedded inside the `new` code block. Once you have the object's unique handle—as assigned to `%handle`, in this case—you can use the object.

Using Objects

To use or control an object, you can use the object's handle to access its properties and functions. If you have an object handle contained in the local variable `%handle`, you can access a property of that object this way:

```
%handle.aproperty = 42;
```

Handles are not the only way to access objects. You can assign objects a name that can be used to access the object, if you don't have a handle at hand. Objects are named using strings, identifiers, or variables containing strings or identifiers. For example, if the object in question is named `MyObject`, all of the following code fragments (A, B, C, D) are the same.

A

```
MyObject.aproperty = 42;
```

B

```
"MyObject".aproperty = 42;
```

C

```
%objname = MyObject;
%objname.aproperty = 42;
```

D

```
%objname = "MyObject";
%objname.aproperty = 42;
```

These examples demonstrate accessing a property field of an object; you invoke object methods (functions) in the same way. Note that the object name—`MyObject`— is a string literal, not a variable. There is no `%` or `$` prefixed to the identifier.

Object Functions

You can call a function referenced through an object this way:

```
%handle.afunction(42, "arg1", "arg2");
```

Note that the function `afunction` can also be referred to as a *method* of the object contained in `%handle`. In the preceding example, the function named `afunction` will be executed. There can be multiple instances of functions named `afunction` in a script, but each must be part of different *namespaces*. The particular instance of `afunction` to be executed will be selected according to the object's namespace and the namespace hierarchy. For more about namespaces, see the sidebar.

Namespaces

Namespaces are means of defining a formal context for variables. Using namespaces allows us to use different variables that have the same name without confusing the game engine, or ourselves.

If you recall the discussion in Chapter 2 about variable scope, you will remember that there are two scopes: global and local. Variables of global scope have a "$" prefix, and variables of local scope have a "%" prefix. Using this notation, we can have two variables—say, `$maxplayers` and `%maxplayers`—that can be used side-by-side, yet whose usage and meaning are completely independent from each other. `%maxplayer` can only be used within a specific function, while `$maxplayer` can be used anywhere in a program. This independence is like having two namespaces.

In fact, `%maxplayer` can be used over and over in different functions, but the values it holds only apply within any given specific function. In these cases, each function is its own *de facto* namespace.

We can arbitrarily assign variables to a namespace by using special prefixes like this:

```
$Game::maxplayers
$Server::maxplayers
```

We can have other variables belonging to the namespace as well:

```
$Game::maxplayers
$Game::timelimit
$Game::maxscores
```

The identifier between the "$" and the "::" can be completely arbitrary—in essence it is a *qualifier*. By qualifying the following variable, it sets a context in which the variable is meaningful.

Just as functions have a de facto namespace (the local scope), objects have their own namespaces. Methods and properties of objects are sometimes called *member functions* and *member variables*. The "member" part refers to the fact that they are members of objects. This membership defines the context, and therefore the namespace, of the methods and properties (member functions and member variables).

So, you can have many different object classes that have properties of the same name, yet they refer only to the objects that belong to that class. You can also have many different instances of an object, and the methods and properties of each instance belong to the individual instance.

In these examples:

```
$myObject.maxSize
$explosion.maxSize
$beast.maxSize
```

the `maxSize` property could have three entirely different meanings. For `$myObject`, `maxSize` might mean the number of items it can carry. For `$explosion`, it might mean how large the blast radius is. For `$beast`, it might mean how tall the creature is.

Team LRN

When an object's function is called, the first parameter is the handle of the object containing the function. Therefore, the function definition of the `afunction` method in the preceding example would actually have four parameters in its parameter list, the first of which will be the `%this` parameter. Note that only the last three parameters are used when you call the `afunction` method. The first parameter that corresponds to the `%this` parameter in the definition is automagically inserted by the engine when you call the function. You may be familiar with the `this` token in C/C++; however, in Torque there is nothing special about it. By prior convention, that variable name is often used when referring to an object's handle within one of its methods, but you could call that parameter anything you want.

If you want to access a field of an object, you always have to use something that evaluates to an object handle or a name followed by a dot followed by the field name, as in the A, B, C, and D code fragments seen earlier. The only exception to this rule is in the sequence of field initialization statements when creating an object with the `new` statement.

Datablocks

A *datablock* is a special kind of object containing a set of characteristics that are used to describe another object's properties. Datablock objects exist simultaneously on the server and all its connected clients. Every copy of a given datablock uses the same handle whether it is on the server or a client.

By convention, datablock identifiers have the form *Name*`Data`. `VehicleData`, `PlayerData`, and `ItemData` are all examples of datablock identifiers. Although datablocks *are* objects, we typically don't explicitly call them objects when referring to them, in order to avoid semantic confusion with regular objects.

A `VehicleData` datablock contains many attributes describing the speed, mass, and other properties that can be applied to a `Vehicle` object. When created, a `Vehicle` object is initialized to reference some already-existing `VehicleData` datablocks that will tell it how to behave. Most objects may come and go throughout the course of the game, but datablocks are created once and are not deleted. Datablocks have their own specific creation syntax:

```
datablock ClassIdentifier(NameIdentifier)
{
      InitializationStatements
};
```

The value of this statement is the handle of the created datablock.

`ClassIdentifier` is an existing datablock class name, like `PlayerData`. `NameIdentifier` is the datablock name you've chosen. In both cases you must use valid identifiers. `InitializationStatements` is a sequence of assignment statements.

The assignment statements assign values to datablock field identifiers. It's possible for the contents of these fields to be accessible by both the script code and the engine code—and in fact that is often the case. In that situation you, of course, need to assign a value to the field that makes sense for the type of information it's supposed to be holding.

You don't have to restrict yourself to only initializing (and later using) fields that are accessible by the engine code. An object can have other fields as well; the engine code can't read them, but the scripts can.

Finally, note that there's a variation on the datablock creation syntax:

```
datablock ClassIdentifier(NameIdentifier : CopySourceIdentifier)
{
      InitializationStatements
};
```

`CopySourceIdentifier` specifies the name of some other datablock from which to copy field values before executing `InitializationStatements`. This other datablock must be of the same class as the datablock you are creating, or a *superclass* of it. This is useful if you want to make a datablock that should be almost exactly like a previously created datablock (with just a few changes) or if you want to centralize the definitions of some characteristics in one datablock that can then be copied by multiple other datablocks.

Game Structure

When you create your game, you can use pretty well any organizational structure you like. Your game will comprise script program modules, graphics images, 3D models, audio files, and various other data definition modules.

The only real limitation in how you structure your game folders is that the *root main module* must reside in the same folder as the Torque Engine executable, and this folder will be the *game root folder*.

The least you should do to sensibly organize your game folders is to have a subtree that contains *common* code, code that would be essentially the same between game types and variations, and another subtree that would contain the *control* code and specific resources that pertain to a particular game, game type, or game variation. GarageGames uses these two basic subtrees, common and control, in its sample games, although the company uses different names (such as *fps*, *rw*, *racing*, and *show*) for variations of the control subtree. See Figure 4.1 for a simple breakdown diagram.

In the game we are creating, we will call the control subtree *control*.

Source files for Torque Script have the .cs extension. After the source files are compiled, they have an extension of .cs.dso. There is no way to convert a .cs.dso file back into a

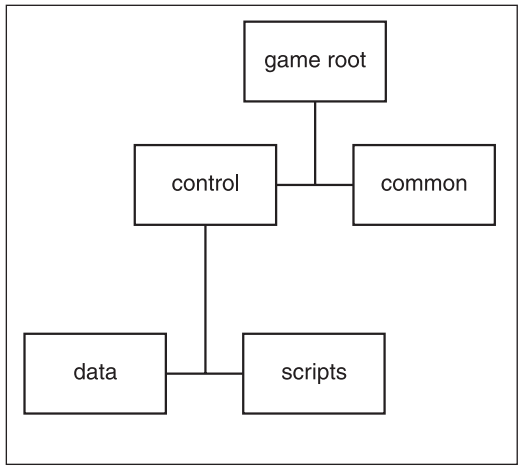


Figure 4.1 General game folder tree.

.cs file, so you must make sure to hang on to your original source files and back them up regularly.

When you launch TGE, it looks for the module main.cs located in the same folder (the *game root folder*, shown below, which shows the general tree format used for the Emaga set of tutorial sample games used in this book) as the TGE executable. In this chapter we will be using a simplified version of this tree. In the distribution of TGE you receive with the CD, the executable is called tge.exe. The particular main.cs file located in the game root folder can be thought of as the *root main module*. This expression is useful for distinguishing that particular main.cs module from others with the same name that aren't in the game root folder.

```
emaga (game root folder)
   common
   client
      debugger
      editor
      help
      lighting
      server
      ui
         cache
   control
      client
         misc
         interfaces
      data
         maps
         models
            avatars
            items
            markers
            weapons
         particles
         sound
         structures
```

```
      docks
      hovels
      towers
server
   misc
   players
   vehicles
   weapons
```

These other main.cs modules are the root modules for the packages in the game. Although it isn't explicitly designated as such, the root main module functions as the root package of the game.

It's important to realize that the folder structure outlined above is not cast in stone. Note that although it is similar, it is still not exactly the same as the format used in the Torque sample games. As long as the root main module is in the same folder as the tge.exe executable, you can use whatever folder structure suits your needs. Of course, you will have to ensure that all of the hard-coded paths in the source modules reflect your customized folder structure.

Figure 4.2 shows the simplified folder tree we will be using for this chapter's sample game, Emaga4. The rectangles indicate folder names, the partial rectangles with the wavy bottoms are source files, and the lozenge shapes indicate binary files. Those items that are not in gray are the items we will be dealing with in this chapter.

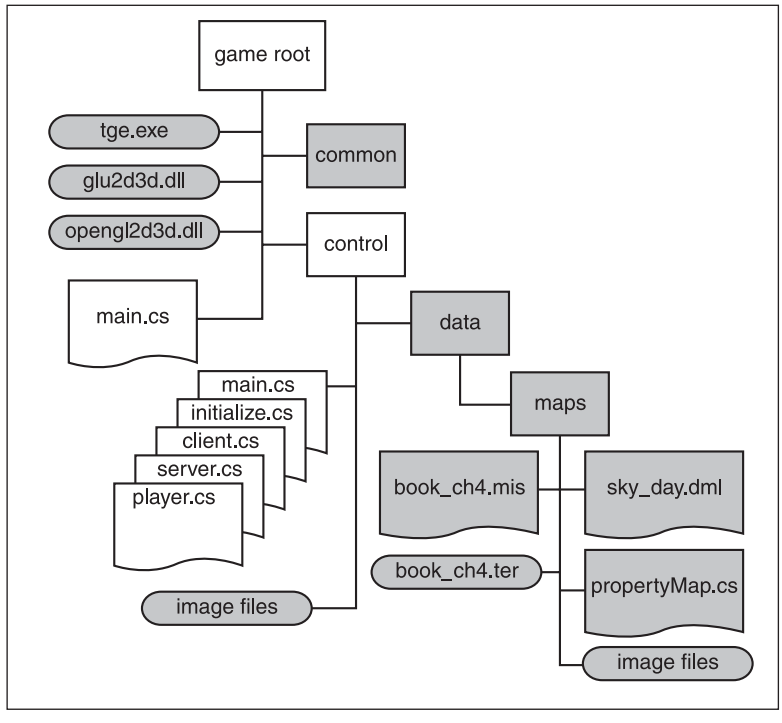


Figure 4.2 The Emaga4 folder tree.

Packages, Add-ons, Mods, and Modules

If you find the terminology confusing, don't fret—it is a little bit less than straightforward at first blush.

The first thing to understand is that the term *Mod* is an abbreviated, or truncated, form of the word *modification*. Mods are changes that people make to existing games, customizing the games to look or play differently. The term is often used in the independent game development scene. The word *Mod* is often capitalized.

What we are doing when we create the Emaga game is in many ways similar to creating a Mod—much like a certain kind of Mod that is often called a *Total Conversion*. Torque, however, is not a game, it is an engine. So we are in reality not modifying an existing game, but, rather, we are creating our own.

Also, there is a bit of an extra wrinkle here: When we create our game, we are going to provide some features that will allow other people to modify our game! To avoid total confusion, we are going to call this capability an *add-on* capability rather than a Mod capability. And we'll refer to the new or extra modules created by other people for our game as *add-ons*.

A *module* is essentially the melding of a program source file in text form with its compiled version. Although we usually refer to the source code version, both the source file version and the compiled (object code, or in the case of Torque, byte code) version are just different forms of the same module.

A *package* is a Torque construct that encapsulates functions that can be dynamically loaded and unloaded during program execution. Scripts often use packages to load and unload the different game types and related functions. Packages can be used to dynamically *overload* functions using the `parent::function()` script mechanism in the packaged function. This is useful for writing scripts that can work with other scripts without any knowledge of those scripts.

To replace the graphical Help features in the Torque demo, for example, you could create one or more source code *modules* that define the new Help features and that together could compose a *Mod* to the graphical Help *package* and that could also be considered a *Mod* to the Torque demo game as a whole.

Clear as mud?

Server versus Client Design Issues

The Torque Engine provides built-in client/server capability. In fact, the engine is designed and built around the client/server model to such a degree that even if you are going to create a single-player game, you will still have both a server side and a client side to your code.

Team LRN

A well-designed online multiplayer game puts as much of the decision-making activity into the hands of the server as possible. This greatly reduces the chances that dishonest players could modify their clients to enable cheating or otherwise gain advantage over other more honest players.

Conversely, a well-designed online multiplayer game only uses the client side to manage the interface with the human player—accepting input, displaying or generating output, and providing setup and game navigation tools.

This emphasis on server-side decisions has the potential to rapidly drain network bandwidth. This can lead to *lag*, a situation where a player's actions are not reflected on the server in a timely fashion. Torque has a highly optimized networking system designed to mitigate against these kinds of problems. For example, most strings of data are transmitted only once between clients and the game server. Anytime a string that has already been transmitted needs to be sent again, a *tag* is sent instead of the full string. The tag is nothing more than a number that identifies the string to be used, so the full string need not be sent again. Another approach is an update *masking* system that allows the engine to only provide updates from the server to its clients of data that has actually changed since the last update.

We will follow these guidelines when designing our sample game.

Common Functionality

The common subtree contains code and resources for the following capabilities:

- Common server functions and utilities, such as authentication
- Common client functions and utilities, such as messaging
- In-game world editor
- Online debugger
- Lighting management and lighting cache control code
- Help features and content files
- User interface definitions, widget definitions, profiles, and images

We will not be using all of these features in the code we'll be looking at in this chapter, but by the end of the book, we *will* be using all of it!

Preparation

In this chapter we will be concentrating on the control scripts found in the control subtree, as outlined in Figure 4.2. To prepare for this, you need to set up your development tree, as follows:

Team LRN

1. In your 3DGPAi1\ RESOURCES folder, locate the EmagaCh4KitInstall.exe program.
2. Run the kit installer. You can install the chapter kit anywhere you like—the default will be to put it in the root folder of your C drive, and this is where I'll assume it is in this book.

You probably won't use more than 15MB of disk space, but you should have the rest available for backups and temporary files and so on.

You will note that there is no main.cs file in the same folder as tge.exe. This is by design, because that is one of the files you will be creating. Also note that there are no .cs files in the control folder either. Again, this is intentional—you will be creating them from this chapter.

The code in Emaga4 is pretty well the bare minimum in terms of the game control code. In later chapters, we will expand on this skeletal implementation as we add more and more useful features and flesh out the game.

Root Main

Once it has found the root main module, Torque compiles it into a special binary version containing *byte code*, a machine-readable format. The game engine then begins executing the instructions in the module. The root package can be used to do anything you like, but the convention established with the GarageGames code is that the root package performs the following functions:

- Performs generic initialization
- Performs the command line parameter parsing and dispatch
- Defines the command line help package
- Invokes packages and add-ons (Mods)

Here is the root main.cs module. Type it in and save it as Emaga4\main.cs. You can skip the comments if you like, in order to minimize your typing.

```
//---------------------------------------------------------------------------
//  ./main.cs
//
//  root main module for 3DGPAI1 emaga4 tutorial game
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//---------------------------------------------------------------------------

// ============================================================================
// ======================== Initializations ===================================
// ============================================================================
```

```
$usageFlag =  false;  //help won't be displayed unless the command line
                      //switch ( -h ) is used

$logModeEnabled =  true; //track the logging state we set in the next line.
SetLogMode(2);   // overwrites existing log file & closes log file at exit.

// =============================================================================
// ======================= Function Definitions ================================
// =============================================================================

function OnExit()
//-----------------------------------------------------------------------------
// This is called from the common code modules. Any last gasp exit
// activities we might want to perform can be put in this function.
// We need to provide a stub to prevent warnings in the log file.
//-----------------------------------------------------------------------------
{
}

function ParseArgs()
//-----------------------------------------------------------------------------
//  handle the command line arguments
//
//  this function is called from the common code
//
//-----------------------------------------------------------------------------
{
  for($i = 1;  $i  < $Game::argc ; $i++) //loop thru all command line args
  {
    $currentarg    = $Game::argv[$i];   // get current arg from the list
    $nextArgument       = $Game::argv[$i+1]; // get arg after the current one
    $nextArgExists = $Game::argc-$i > 1;// if there *is* a next arg, note that
    $logModeEnabled =  false;            // turn this off; let the args dictate
                                  // if logging should be enabled.

    switch$($currentarg)
    {
      case "-?":  // the user wants command line help, so this causes the
        $usageFlag = true;   // Usage function to be run, instead of the game
        $argumentFlag[$i] = true;                // adjust the argument count

      case "-h":          // exactly the same as "-?"
```

```
         $usageFlag = true;
         $argumentFlag[$i] = true;
     }
   }
}

function Usage()
//-----------------------------------------------------------------------------
// Display the command line usage help
//-----------------------------------------------------------------------------
{
//  NOTE: any logging entries are written to the file 'console.log'
  Echo("\n\nemaga4 command line options:\n\n" @
          " -h, -?               display this message\n" );
}

function  LoadAddOns(%list)
//-----------------------------------------------------------------------------
// Exec each of the startup scripts for add-ons.
//-----------------------------------------------------------------------------
{
  if (%list $= "")
    return;
  %list = NextToken(%list, token, ";");
  LoadAddOns(%list);
  Exec(%token @  "/main.cs");
}

// ============================================================================
// ================ Module Body - Inline Statements ========================
// ============================================================================
//  Parse the command line arguments
ParseArgs();

//  Either display the help message or start the program.
if  ($usageFlag)
{
  EnableWinConsole(true);// send logging output to a Windows console window
  Usage();
  EnableWinConsole(false);
  Quit();
}
```

```
else
{

  //  scan argument list, and log an Error message for each unused argument
  for ($i = 1;  $i  < $Game::argc; $i++)
  {
     if (!$argumentFlag[$i])
        Error("Error: Unknown command line argument:  " @ $Game::argv[$i]);
  }

  if  (!$logModeEnabled)
  {
     SetLogMode(6);       //  Default to a new log file each session.
  }
  //  Set the add-on path list to specify the folders that will be
  //  available to the scripts and engine. Note that *all* required
  //  folder trees are included: common and control as well as the
  //  user add-ons.
  $pathList = $addonList !$= "" ? $addonList @ ";control;common" : "control;common";
  SetModPaths($pathList);

  // Execute startup script for the common code modules
  Exec("common/main.cs");

  // Execute startup script for the control specific code modules
  Exec("control/main.cs");

  // Execute startup scripts for all user add-ons
  Echo("--------- Loading Add-ons ---------");
  LoadAddOns($addonList);
  Echo("Engine initialization complete.");

  OnStart();
}
```

This is a fairly robust root main module. Let's take a closer look at it.

In the Initializations section, the `$usageFlag` variable is used to trigger a simple Help display for command line use of tge.exe. It is set to `false` here; if the user specifies the `-?` or `-h` flags on the command line, then this flag will be set to `false`.

After the usage flag, we set the log mode and enable logging. Logging allows us to track what is happening within the code. When we use the `Echo()`, `Warn()`, or `Error()` functions, their output is sent to the console.log file, in the root game folder.

Team LRN

The stub routine `OnExit()` is next. A *stub routine* is a function that is defined but actually does nothing. The common code modules have a call to this routine, but we have nothing for it to do. We could just leave it out, but a good policy is to provide an empty stub to avoid warning messages from appearing in our log file—when the Torque Engine tries to call a nonexistent function, it generates a warning.

Then there is the `ParseArgs()` function. Its job is to step through the list of command line arguments, or parameters, and perform whatever tasks you want based upon what arguments the user provided. In this case we'll just include code to provide a bare-bones usage, or Help, display.

Next is the actual `Usage()` function that displays the Help information.

This is followed by the `LoadAddOns()` routine. Its purpose is to walk through the list of add-ons specified by the user on the command line and to load the code for each. In Emaga4 there is no way for the user to specify add-ons or Mods, but (you knew there was a *but* coming, didn't you?) we still need this function, because we treat our common and control modules as if they were add-ons. They are always added to the list in such a way that they get loaded first. So this function is here to look after them.

After the function definitions we move into the in-line program statements. These statements are executed at load time—when the module is loaded into memory with the `Exec()` statement. When Torque runs, after the engine gets itself sorted out, it always loads the root main module (this module) with an `Exec()` statement. All of the other script modules are loaded as a result of what this module does.

The first thing that happens is a call to the `ParseArgs()` function, which we saw earlier. It sets the `$usageFlag` variable for us, you will recall.

Next is the block of code that examines the `$usageFlag` and decides what to do: either display the usage Help information or continue to run the game program. If we are not displaying the usage information, we move into the code block after the `else`.

The first thing we do in here is check to see if there are any unused arguments from the command line. If there are, that means the program doesn't understand the arguments and there was some kind of error, which we indicate with the `Error()` function and a useful message.

After that we set the log mode, if logging has been enabled.

Next, we build the lists that help Torque find our add-ons. We notify Torque about the required folder paths by passing the list to the `SetModPaths()` function.

Then we call the main module for the common code. This will proceed to load all the required common modules into memory, initialize the common functions, and basically get the ball rolling over there. We will talk about the common code modules in a later chapter.

Team LRN

After that we do the same thing for the control code modules, the details of which we will cover later in this chapter.

Then we actually start loading the add-ons using the previously defined `LoadAddOns()` function.

Finally, we make a call to `OnStart()`. This will call all versions of `OnStart()` that appear in the add-on packages in order of their appearance in `$addonList`, with common being first, control next, and finally this root main module. If there is an `OnStart()` defined in common, then it gets called. Then the one in control, and so on.

When we get to the end of the module, the various threads initiated by the `OnStart()` calls are ticking over, doing their own things.

So now what? Well, our next point of interest is the control/main.cs module, which we called with the `Exec()` function just before we started loading the add-ons.

Control Main

The main.cs module for the control code is next on our tour. Its primary purposes in Emaga4 are to define the control package and to call the control code initialization functions. (In later chapters we will expand on the role of this module.) Following is the control/main.cs module. Type it in and save it as Emaga4\control\main.cs.

```
//---------------------------------------------------------------------------
// control/main.cs
//  main control module for 3DGPAI1 emaga4 tutorial game
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//---------------------------------------------------------------------------
//
//---------------------------------------------------------------------------
// Load up defaults console values.

// Defaults console values

//---------------------------------------------------------------------------
// Package overrides to initialize the mod.
package control {

function OnStart()
//---------------------------------------------------------------------------
// Called by root main when package is loaded
//---------------------------------------------------------------------------
```

```
{
   Parent::OnStart();
   Echo("\n--------- Initializing control module ---------");

   // The following scripts contain the preparation code for
   // both the client and server code. A client can also host
   // games, so they need to be able to act as servers if the
   // user wants to host a game. That means we always prepare
   // to be a server at anytime, unless we are launched as a
   // dedicated server.
   Exec("./initialize.cs");
   InitializeServer(); // Prepare the server-specific aspects
   InitializeClient(); // Prepare the client-specific aspects
}

function OnExit()
//---------------------------------------------------------------------------
// Called by root main when package is unloaded
//---------------------------------------------------------------------------
{

   Parent::onExit();
}

}; // Client package
ActivatePackage(control); // Tell TGE to make the client package active
```

Not a whole lot happens in here at the moment, but it is a necessary module because it defines our control package.

First, the *parent* `OnStart()` function is called. This would be the version that resides in root main, which we can see doesn't have anything to do.

Then the initialize.cs module is loaded, after which the two initialization functions are called.

Finally, there is the `OnExit()` function, which does nothing more than pass the buck to the `OnExit()` function in the root main module.

All in all, control/main.cs is a fairly lazy, though important, little module.

Debugging Scripts Using the trace() Function

The engine adds extra commentary to the log file. Extremely useful are the notations that tell you when the engine execution has just begun executing in a particular function or is just about to leave a particular function. The trace lines include the values of any arguments used when the function is entered and the contents of the return value when leaving a function.

Here is a fragmentary example of what the trace output can look like:

```
Entering GameConnection::InitialControlSet(1207)
Setting Initial Control Object
    Entering Editor::checkActiveLoadDone()
    Leaving Editor::checkActiveLoadDone - return 0
    Entering GuiCanvas::setContent(Canvas, PlayGui)
        Entering PlayGui::onWake(1195)
    Activating DirectInput...
    keyboard0 input device acquired.
        Leaving PlayGui::onWake - return
        Entering GuiCanvas::checkCursor(Canvas)
            Entering (null)::cursorOff()
            Leaving (null)::cursorOff - return
        Leaving GuiCanvas::checkCursor - return
    Leaving GuiCanvas::setContent - return
Leaving GameConnection::InitialControlSet - return
Entering (null)::DoYaw(-9)
Leaving (null)::DoYaw - return -0.18
Entering (null)::DoPitch(7)
Leaving (null)::DoPitch - return 0.14
Entering (null)::DoYaw(-6)
```

To turn on the `trace` function, add the following statement to the first line of your root main.cs file:

```
trace(true);
```

To turn off the `trace` function, insert this statement at the place in the code where you would like to turn tracing off:

```
Trace(false);
```

Initialization

The control/initialize.cs module will, in later chapters, become two different modules—one for the server code and one for the client code. Right now, we have a fairly limited amount of work to do, so we'll just house the initialization functions for the two ends in

the same module. Here is the control/initialize.cs module. Type it in and save it as Emaga4\control\initialize.cs.

```
//=============================================================================
// control/initialize.cs
//
//  control initialization module for 3DGPAI1 emaga4 tutorial game
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//=============================================================================


function InitializeServer()
//-----------------------------------------------------------------------------
// Prepare some global server information & load the game-specific module
//-----------------------------------------------------------------------------
{
   Echo("\n--------- Initializing module: emaga server ---------");

   // Specify where the mission files are.
   $Server::MissionFileSpec = "*/missions/*.mis";

   InitBaseServer(); // basic server features defined in the common modules

   // Load up game server support script
   Exec("./server.cs");

   createServer("SinglePlayer", "control/data/maps/book_ch4.mis");
}


function InitializeClient()
//-----------------------------------------------------------------------------
// Prepare some global client information, fire up the graphics engine,
// and then connect to the server code that is already running in another
// thread.
//-----------------------------------------------------------------------------
{
   Echo("\n--------- Initializing module: emaga client ---------");

   InitBaseClient(); // basic client features defined in the common modules

  // these are necessary graphics settings
```

```
   $pref::Video::allowOpenGL   = true;
   $pref::Video::displayDevice = "OpenGL";

    // Make sure a canvas has been built before any gui scripts are
    // executed because many of the controls depend on the canvas to
    // already exist when they are loaded.

    InitCanvas("Egama4 - 3DGPAi1 Sample Game"); // Start the graphics system.

    Exec("./client.cs");

    %conn = new GameConnection(ServerConnection);
    %conn.connectLocal();
}
```

First is the `InitializeServer()` function. This is where we set up a global variable that indicates to the game engine the folder tree where the *map* (also called *mission*) files will be located.

Next, we prepare the server for operation by performing the common code initialization using the `InitBaseServer()` function. This allows us to get the server code running full-bore, which we can do using the `createServer()` call. We tell the function that this will be a single-player game and that we are going to load up the map control/data/maps/book_ch4.mis.

After that, we load the module that contains the game code, which is server-side code.

Then we do the client-side initialization in the `InitializeClient()` function. This is a bit more involved. After performing the common code initialization with `InitBaseClient()`, we set up some global variables that the engine uses to prepare the graphics system for start-up.

And that happens with the `InitCanvas()` call. The parameter we pass in is a string that specifies the name of the window that the game will be running in.

Then we load the control/client.cs module, which we'll cover next in this chapter.

We're getting warm now!

Next, we create a connection object using the `GameConnection()` function. This gives us an object that we will use from now on when referring to the connection.

Now we use that connection object to connect to the server using a local connection. We don't ever actually use the network or any network ports.

Client

The control/client.cs module is chock-full of good stuff. This is another module that will need to have some of its code divested when it grows in later chapters. The main activities taking place in here are as follows:

- Creation of a key map with key bindings
- Definition of a callback that gets called from with Torque to generate a 3D view
- Definition of an interface to hold the 3D view
- Definition of a series of functions that hook key commands to avatar motion
- A series of stub routines

Here is the control/client.cs module. Type it in and save it as Emaga4\control\client.cs.

```
//============================================================================
// control/client.cs
//
// This module contains client specific code for handling
// the setup and operation of the player's in-game interface.
//
// 3DGPAI1 emaga4 tutorial game
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//============================================================================

if ( IsObject( playerKeymap ) )  // If we already have a player key map,
   playerKeymap.delete();         // delete it so that we can make a new one
new ActionMap(playerKeymap);

$movementSpeed = 1;              // m/s   for use by movement functions

//----------------------------------------------------------------------------
// The player sees the game via this control
//----------------------------------------------------------------------------
new GameTSCtrl(PlayerInterface) {
   profile = "GuiContentProfile";
   noCursor = "1";
};

function PlayerInterface::onWake(%this)
//----------------------------------------------------------------------------
// When PlayerInterface is activated, this function is called.
//----------------------------------------------------------------------------
```

```
{
   $enableDirectInput = "1";
   activateDirectInput();

   // restore the player's key mappings
   playerKeymap.push();
}

function GameConnection::InitialControlSet(%this)
//-----------------------------------------------------------------------------
// This callback is called directly from inside the Torque Engine
// during server initialization.
//-----------------------------------------------------------------------------
{
   Echo ("Setting Initial Control Object");

   // The first control object has been set by the server
   // and we are now ready to go.

   Canvas.SetContent(PlayerInterface);
}

//============================================================================
// Motion Functions
//============================================================================

function GoLeft(%val)
//-----------------------------------------------------------------------------
// "strafing"
//-----------------------------------------------------------------------------
{
   $mvLeftAction = %val;
}

function GoRight(%val)
//-----------------------------------------------------------------------------
// "strafing"
//-----------------------------------------------------------------------------
{
   $mvRightAction = %val;
}
```

```
function GoAhead(%val)
//-----------------------------------------------------------------------------
// running forward
//-----------------------------------------------------------------------------
{
   $mvForwardAction = %val;
}

function BackUp(%val)
//-----------------------------------------------------------------------------
// running backwards
//-----------------------------------------------------------------------------
{
   $mvBackwardAction = %val;
}

function DoYaw(%val)
//-----------------------------------------------------------------------------
// looking, spinning or aiming horizontally by mouse or joystick control
//-----------------------------------------------------------------------------
{
   $mvYaw += %val * ($cameraFov / 90) * 0.02;
}

function DoPitch(%val)
//-----------------------------------------------------------------------------
// looking vertically by mouse or joystick control
//-----------------------------------------------------------------------------
{
   $mvPitch += %val * ($cameraFov / 90) * 0.02;
}

function DoJump(%val)
//-----------------------------------------------------------------------------
// momentary upward movement, with character animation
//-----------------------------------------------------------------------------
{
   $mvTriggerCount2++;
}

//============================================================================
// View Functions
```

```
//============================================================================

function Toggle3rdPPOVLook( %val )
//----------------------------------------------------------------------------
// Enable the "free look" feature. As long as the mapped key is pressed,
// the player can view his avatar by moving the mouse around.
//----------------------------------------------------------------------------
{
   if ( %val )
      $mvFreeLook = true;
   else
      $mvFreeLook = false;
}

function Toggle1stPPOV(%val)
//----------------------------------------------------------------------------
// switch between 1st and 3rd person point-of-views.
//----------------------------------------------------------------------------
{
   if (%val)
   {
      $firstPerson = !$firstPerson;
   }
}

//============================================================================
// keyboard control mappings
//============================================================================
// these ones available when player is in game
playerKeymap.Bind(keyboard, up, GoAhead);
playerKeymap.Bind(keyboard, down, BackUp);
playerKeymap.Bind(keyboard, left, GoLeft);
playerKeymap.Bind(keyboard, right, GoRight);
playerKeymap.Bind( keyboard, numpad0, DoJump );
playerKeymap.Bind( mouse, xaxis, DoYaw );
playerKeymap.Bind( mouse, yaxis, DoPitch );
playerKeymap.Bind( keyboard, z, Toggle3rdPPOVLook );
playerKeymap.Bind( keyboard, tab, Toggle1stPPOV );

// these ones are always available
GlobalActionMap.BindCmd(keyboard, escape, "", "quit();");
GlobalActionMap.Bind(keyboard, tilde, ToggleConsole);
```

```
//============================================================================
// The following functions are called from the client common code modules.
// These stubs are added here to prevent warning messages from cluttering
// up the log file.
//============================================================================
function onServerMessage()
{
}
function onMissionDownloadPhase1()
{
}
function onPhase1Progress()
{
}
function onPhase1Complete()
{
}
function onMissionDownloadPhase2()
{
}
function onPhase2Progress()
{
}
function onPhase2Complete()
{
}
function onPhase3Complete()
{
}
function onMissionDownloadComplete()
{
}
```

Right off the bat, a new `ActionMap` called `playerKeymap` is created. This is a structure that holds the mapping of key commands to functions that will be performed—a mechanism often called *key binding*, or *key mapping*. We create the new `ActionMap` with the intent to populate it later in the module.

Then we define the 3D control (TS, or *ThreeSpace*) we call `PlayerInterface` (because that's what it is), which will contain our view into the 3D world. It's not a complex definition. It basically uses a profile defined in the common code—something we'll explore in a later chapter. If we want to use our mouse to provide view manipulation, we must set the `noCursor` property of the control to 1, or `true`.

Team LRN

Then we define a method for the `PlayerInterface` control that describes what to do when the control becomes active ("wakes up"). It's not much, but what it does is activate `DirectInput` in order to grab any user inputs at the keyboard or mouse and then make the `playerKeymap` bindings active.

Next, we define a callback method for the `GameConnection` object (you know, the one we created back there in control/main.cs). The engine invokes this method internally when the server has established the connection and is ready to hand control over to us. In this method we assign our player interface control to the `Canvas` we created earlier in the `InitializeClient()` function in the control/initialize.cs module.

After that, we define a whole raft of motion functions to which we will later bind keys. Notice that they employ global variables, such as `$mvLeftAction`. This variable and others like it, each of which starts with `$mv`, are seen and used internally by the engine.

Then there is a list of key bindings. Notice that there are several variations of the `Bind` calls. First, there are binds to our `playerKeymap`, which makes sense. Then there are binds to the `GlobalActionMap`; these bindings are available at all times when the program is running, not just when an actual game simulation is under way, which is the case with a normal action map.

Finally, there is a list of stub routines. All of these routines are called from within the common code package. We don't need them to do anything yet, but as before, in order to minimize log file warnings, we create stub routines for the functions.

Server

The control/server.cs module is where game-specific server code is located. Most of the functionality that is carried in this module is found in the form of methods for the GameConnection class. Here is the control/server.cs module. Type it in and save it as Emaga4\control\server.cs.

```
//==============================================================================
// control/server.cs
//
//  server-side game specific module for 3DGPAI1 emaga4 tutorial game
//  provides client connection management and player/avatar spawning
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//==============================================================================
function OnServerCreated()
//------------------------------------------------------------------------------
// Once the engine has fired up the server, this function is called
//------------------------------------------------------------------------------
{
```

```
   Exec("./player.cs"); // Load the player datablocks and methods
}

//=============================================================================
// GameConnection Methods
// Extensions to the GameConnection class. Here we add some methods
// to handle player spawning and creation.
//=============================================================================

function GameConnection::OnClientEnterGame(%this)
//-----------------------------------------------------------------------------
// Called when the client has been accepted into the game by the server.
//-----------------------------------------------------------------------------
{
   // Create a player object.
   %this.spawnPlayer();
}

function GameConnection::SpawnPlayer(%this)
//-----------------------------------------------------------------------------
// This is where we place the player spawn decision code.
// It might also call a function that would figure out the spawn
// point transforms by looking up spawn markers.
// Once we know where the player will spawn, then we create the avatar.
//-----------------------------------------------------------------------------
{

   %this.createPlayer("0 0 220 1 0 0 0");
}
function GameConnection::CreatePlayer(%this, %spawnPoint)
//-----------------------------------------------------------------------------
// Create the player's avatar object, set it up, and give the player control
// of it.
//-----------------------------------------------------------------------------
{
   if (%this.player > 0)//The player should NOT already have an avatar object.
   {                       // If he does, that's a Bad Thing.
      Error( "Attempting to create an angus ghost!" );
   }

   // Create the player object
   %player = new Player() {
```

```
      dataBlock = HumanMaleAvatar;   // defined in player.cs
      client = %this;            // the avatar will have a pointer to its
   };                             // owner's connection

   // Player setup...
   %player.setTransform(%spawnPoint); // where to put it

   // Give the client control of the player
   %this.player = %player;
   %this.setControlObject(%player);
}
//=============================================================================
// The following functions are called from the server common code modules.
// These stubs are added here to prevent warning messages from cluttering
// up the log file.
//=============================================================================
function ClearCenterPrintAll()
{
}
function ClearBottomPrintAll()
{
}
```

The first function, `OnServerCreated`, manages what happens immediately after the server is up and running. In our case we need the player-avatar datablocks and methods to be loaded up so they can be transmitted to the client.

Then we define some `GameConnection` methods. The first one, `OnClientEnterGame`, simply calls the `SpawnPlayer` method, which then calls the `CreatePlayer` method using the hard-coded transform provided.

`CreatePlayer` then creates a new player object using the player datablock defined in control/player.cs (which we will review shortly). It then applies the transform (which we created manually earlier) to the player's avatar and then transfers control to the player.

Finally, there are a couple more stub routines. That's the end of them—for now—I promise!

Player

The control/player.cs module defines the player datablock and methods for use by this datablock for various things. The datablock will use the standard male model, which in this case has been named player.dts. Figure 4.3 shows the standard male avatar in the Emaga4 game world.

Figure 4.3 Player-avatar in Emaga4.

Here is the control/player.cs module. Type it in and save it as Emaga4\control\player.cs.

```
//-----------------------------------------------------------------------
// control/player.cs
//
//  player definition module for 3DGPAI1 emaga4 tutorial game
//
//  Copyright (c) 2003 by Kenneth C. Finney.
//-----------------------------------------------------------------------
datablock PlayerData(HumanMaleAvatar)
{
   className = Avatar;
   shapeFile = "~/player.dts";
   emap = true;
   renderFirstPerson = false;
   cameraMaxDist = 4;
   mass = 100;
   density = 10;
   drag = 0.1;
   maxdrag = 0.5;
   maxEnergy =  100;
   maxDamage = 100;
   maxForwardSpeed = 15;
   maxBackwardSpeed = 10;
   maxSideSpeed = 12;
```

```
   minJumpSpeed = 20;
   maxJumpSpeed = 30;
   runForce = 4000;
   jumpForce = 1000;
   runSurfaceAngle  = 70;
   jumpSurfaceAngle = 80;
};


//----------------------------------------------------------------------------
// Avatar Datablock methods
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------

function Avatar::onAdd(%this,%obj)
{
}

function Avatar::onRemove(%this, %obj)
{
   if (%obj.client.player == %obj)
      %obj.client.player = 0;
}
```

The datablock used is the PlayerData class. It is piled to the gunwales with useful stuff. Table 4.2 provides a summary description of each of the properties.

There are many more properties available for the avatar, which we aren't using right now. We can also define our own properties for the datablock and access them, through an instance object of this datablock, from anywhere in the scripts.

Last but not least, there are two methods defined for the datablock. The two basically define what happens when we add a datablock and when we remove it. We will encounter many others in later chapters.

Running Emaga4

Once you've typed in all of the modules, you should be in a good position to test Emaga4. Emaga4 is a fairly minimalist program. When you launch tge.exe, you will be deposited directly into the game. Once you have been deposited in the game, you have a small set of keyboard commands available to control your avatar, as shown in Table 4.3.

Table 4.2 Emaga4 Avatar Properties

Property	Description
`className`	Defines an arbitrary class that the avatar can belong to.
`shapeFile`	Specifies the file that contains the 3D model of the avatar.
`emap`	Enables environment mapping on the avatar model.
`renderFirstPerson`	When true, causes the avatar model to be visible when in first-person point-of-view mode.
`cameraMaxDist`	Maximum distance from the avatar for the camera in third-person point-of-view mode.
`mass`	The mass of the avatar in terms of the game world.
`density`	Arbitrarily defined density.
`drag`	Slows down the avatar through simulated friction.
`maxdrag`	Maximum allowable drag.
`maxEnergy`	Maximum energy allowed.
`maxDamage`	Maximum damage points that can be sustained before the avatar is killed.
`maxForwardSpeed`	Maximum speed allowable when moving forward.
`maxBackwardSpeed`	Maximum speed allowable when moving backward.
`maxSideSpeed`	Maximum speed allowable when moving sideways (strafing).
`minJumpSpeed`	Below this speed, you can't make the avatar jump.
`maxJumpSpeed`	Above this speed, you can't make the avatar jump.
`jumpForce`	The force, and therefore the acceleration, when jumping.
`runForce`	The force, and therefore the acceleration, when starting to run.
`runSurfaceAngle`	Maximum slope (in degrees) that the avatar can run on.
`jumpSurfaceAngle`	Maximum slope (in degrees) that the avatar can jump on, usually somewhat less than `runSurfaceAngle`.

Table 4.3 Emaga4 Navigation Keys

Key	Description
Up Arrow	Run forward
Down Arrow	Run backward
Left Arrow	Run (strafe) left
Right Arrow	Run (strafe) right
Numpad 0	Jump
z	Free look (hold key and move mouse)
Tab	Toggle player point of view
Escape	Quit game
Tilde	Open console

Team LRN

After you have created all of the modules, you can run Emaga4 simply by double-clicking on Emaga4\tge.exe. You will "spawn" in to the game world above the ground, and drop down. When you hit the ground, your view will shake from the impact. If you turn your player around, using the mouse, you will see the view shown in Figure 4.4.

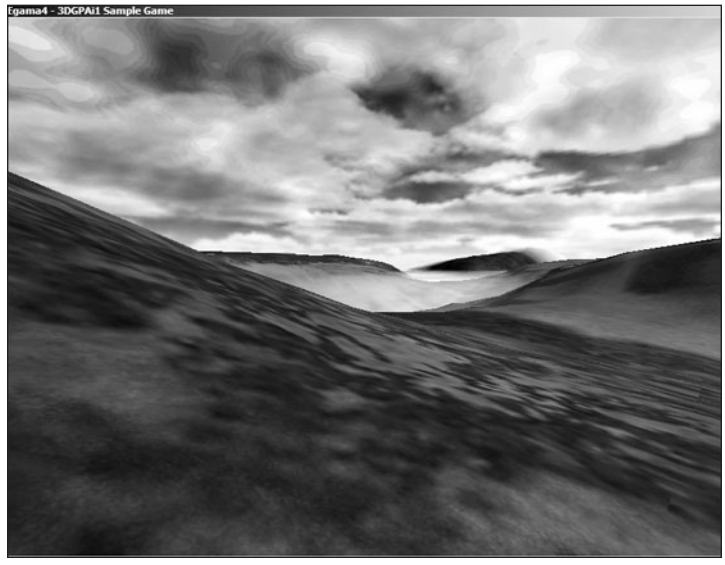


Figure 4.4 Looking around the Emaga4 game world.

After spawning, you can run around the countryside, admire your avatar with the Tab and z keys, and jump.

Moving Right Along

You should have a fairly simple game now. I'll be the first to admit that there is not much to do within the game, but then that wasn't the point, really. By stripping down to a bare-bones code set, we get a clearer picture of what takes place in our script modules.

By typing in the code presented in this chapter, you should have added the following files in your emaga4 folder:

C:\emaga4\main.cs

C:\emaga4\control\main.cs

C:\emaga4\control\client.cs

C:\emaga4\control\server.cs

C:\emaga4\control\initialize.cs

C:\emaga4\control\player.cs

The program you have will serve as a fine skeleton program upon which you can build *your* game in the manner that *you* want.

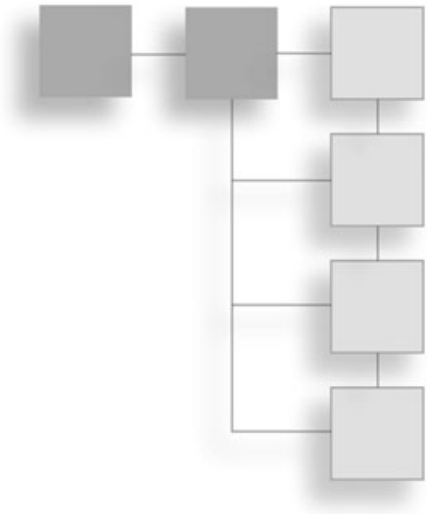
By creating it, you've seen how the responsibilities of the client and server portions of the game are divvied out.

You've also learned that your player's avatar needs to have a programmatic representation in the game that describes the characteristics of the avatar, and how it does things.

In the next chapter we will expand the game by adding game play code on both the client and the server sides.

Team LRN

CHAPTER 5



Game Play

In Chapter 4 we created a small game, Emaga4. Well, not really a game—more of a really simple virtual reality simulation. We created a few important modules to get the ball rolling.

In this chapter we'll build on that humble beginning and grow toward something with some game play challenge in it, called Emaga5. There will be some tasks to perform (*goals*) and some things to make those tasks just that much harder (*dramatic tension*).

To make this happen we'll have to add a fair number of new control modules, modify some of the existing ones, and reorganize the folder tree somewhat. We'll do that in reverse order, starting with the reorganization.

The Changes

You will recall that there are two key branches in the folder tree: common and control. As before, we won't worry about the common branch.

Folders

The control branch contained all of our code in the Chapter 4 version. For this chapter we'll use a more sophisticated structure. When you run the EmagaCh5KitInstall program, it will automatically create the new folder tree for you. It's important for you to become familiar with it, so study Figure 5.1 for a few minutes.

After examining Figure 5.1, take a few moments to run the EmagaCh5KitInstall program. You will find it in the 3DGPAi1\RESOURCES folder. After it does its business, it will have installed everything except the key modules that we're going to explore in detail. There is still some manual assembly involved.

Team LRN

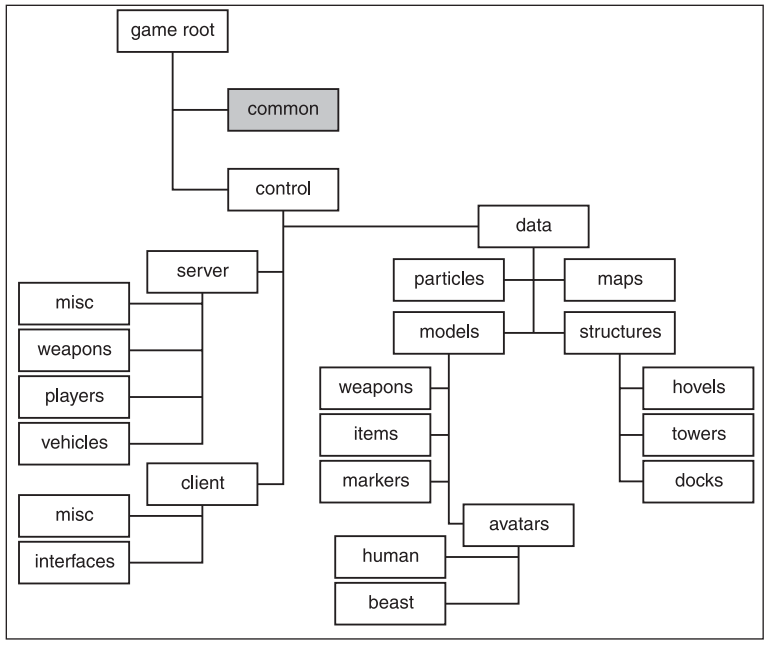


Figure 5.1 The Emaga5 folder tree.

The new folder tree is the one we will be sticking with for the rest of the book. We will be adding a couple more folder nodes for specialized features in later chapters, but otherwise, this is the final form.

Modules

You will not need to type in the root main module again, because it won't be any different this time around.

In the control branch, the first major difference is that the initialize.cs module has been split in two, with a client version and a server version. Each of the new modules is now located in its respective branches—control/server/ and control/client/. They still perform the same tasks as before, but splitting the initialize functions and putting them in their permanent homes prepares us for all our later organizational needs.

There were also the two modules: control/server.cs and control/client.cs. We will now expand these and relocate them as control/server/server.cs and control/client/client.cs, respectively.

The final module from Chapter 4 is player.cs. We will be expanding it greatly and relocating it to control/server/players/player.cs.

Furthermore, we will add several new modules to handle various functional features of the game. We'll address each file as we encounter it in the chapter.

Make sure you have run the EmagaCh5KitInstall program before proceeding, because it creates our folder tree for us.

Control Modules

As before, the control modules are where we focus our game-specific energies. In the root control folder is the control main module. The rest of the code modules are divided

between the client and server branches. The data branch is where our art and other data definition resources reside.

control/main.cs

Type in the following code and save it as the control main module at C:\Emaga5\control\main.cs. In order to save on space, there are fewer source code comments than in the last chapter.

```
//---------------------------------------------------------------------------
// control/main.cs
//  Copyright (c) 2003 by Kenneth C.  Finney.
//---------------------------------------------------------------------------
Exec("./client/presets.cs");
Exec("./server/presets.cs");

package control {
function OnStart()
{
   Parent::OnStart();
   Echo("\n++++++++++++ Initializing control module ++++++++++++");
   Exec("./client/initialize.cs");
   Exec("./server/initialize.cs");
   InitializeServer(); // Prepare the server-specific aspects
   InitializeClient(); // Prepare the client-specific aspects
}
function OnExit()
{
   Parent::onExit();
}
}; // Client package
ActivatePackage(control); // Tell TGE to make the client package active
```

Right off the bat, we can see some new additions. The two `Exec` statements at the beginning load two files that contain *presets*. These are script variable assignment statements. We make these assignments here to specify standard or default settings. Some of the variables in those files pertain to graphics settings, others specify input modes, and things like that.

Next we have the control package, which has a few minor changes in its `OnStart()` function. This is where we load the two new initialization modules and then call the initialization functions for the server and then the client.

Client Control Modules

Modules that affect only the client side of the game are contained in the control/client folder tree. The client-specific activities deal with functions like the interface screens and displays, user input, and coordinating game start-up with the server side of the game.

control/client/client.cs

Many features that were in client.cs in the last chapter are now found in other modules. The key mapping and interface screen code that were located in this module, client.cs, have been given homes of their own, as you'll see later. Type in the following code and save it as C:\Emaga5\control\client\client.cs.

```
//============================================================================
// control/client/client.cs
//  Copyright (c) 2003 by Kenneth C.  Finney.
//============================================================================
function LaunchGame()
{
   createServer("SinglePlayer", "control/data/maps/book_ch5.mis");
   %conn = new GameConnection(ServerConnection);
   %conn.setConnectArgs("Reader");
   %conn.connectLocal();
}
function ShowMenuScreen()
{
   // Start up the client with the menu...
   Canvas.setContent( MenuScreen );
   Canvas.setCursor("DefaultCursor");
}
function SplashScreenInputCtrl::onInputEvent(%this, %dev, %evt, %make)
{
   if(%make)
   {
     ShowMenuScreen();
   }
}
//============================================================================
// stubs
//============================================================================
function onServerMessage()
{
}
```

```
function onMissionDownloadPhase1()
{
}
function onPhase1Progress()
{
}
function onPhase1Complete()
{
}
function onMissionDownloadPhase2()
{
}
function onPhase2Progress()
{
}
function onPhase2Complete()
{
}
function onPhase3Complete()
{
}
function onMissionDownloadComplete()
{
}
```

We've added three new functions, the first of which is `LaunchGame()`. The code contained should be familiar from Emaga4. This function is executed when the user clicks on the Start Game button on the front menu screen of the game—the other options available on the front screen are Setup and Quit.

Next is `ShowMenuScreen()`, which is invoked when the user clicks the mouse or hits a key when sitting viewing the splash screen. The code it invokes is also familiar from Emaga4.

The third function, `SplashScreenInputCtrl::onInputEvent()`, is a callback method used by a `GuiInputControl`, in this case the `SplashScreenInputCtrl`, which is attached to the splash screen for the narrow purpose of simply waiting for user input, and when it happens, closing the splash screen. We get the user input value in the `%make` parameter. Figure 5.2 shows what the splash screen looks like.

The rest of the functions are the by-now-famous stub routines. These are mostly client/server mission (map) loading and coordination functions. These will get more attention in later chapters. You are free to leave out the stub routines, but if you do, you will end up with a ton of warning messages in the log file.

Figure 5.2 The Emaga5 splash screen.

control/client/interfaces-/menuscreen.gui

All of the user interface and display screens now have modules of their own, and they reside in the interfaces branch of the client tree. Note that the extension of these modules is .gui. Functionally, a .gui is the same as a .cs source module. They both can contain any kind of valid script code, and both compile to the .dso binary format. Type in the following code and save it as C:\Emaga5\control\client\interfaces\menuscreen.gui.

```
new GuiChunkedBitmapCtrl(MenuScreen) {
   profile = "GuiContentProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   bitmap = "./interfaces/emaga_background";
   useVariable = "0";
   tile = "0";
   new GuiButtonCtrl() {
      profile = "GuiButtonProfile";
      horizSizing = "right";
      vertSizing = "top";
      position = "29 300";
      extent = "110 20";
      minExtent = "8 8";
      visible = "1";
      command = "LaunchGame();";
      helpTag = "0";
      text = "Start Game";
      groupNum = "-1";
      buttonType = "PushButton";
```

```
   };
   new GuiButtonCtrl() {
      profile = "GuiButtonProfile";
      horizSizing = "right";
      vertSizing = "top";
      position = "29 350";
      extent = "110 20";
      minExtent = "8 8";
      visible = "1";
      command = "Canvas.pushDialog(SetupScreen);";
      helpTag = "0";
      text = "Setup";
      groupNum = "-1";
      buttonType = "PushButton";
   };
   new GuiButtonCtrl() {
      profile = "GuiButtonProfile";
      horizSizing = "right";
      vertSizing = "top";
      position = "29 400";
      extent = "110 20";
      minExtent = "8 8";
      visible = "1";
      command = "Quit();";
      helpTag = "0";
      text = "Quit";
      groupNum = "-1";
      buttonType = "PushButton";
   };
};
```

What we have here is a hierarchical definition of nested objects. The outer object that contains the others is the `MenuScreen` itself, defined as a `GuiChunkedBitmapCtrl`. Many video cards have texture size limits; for some nothing over 512 pixels by 512 pixels can be used. The `ChunkedBitmap` splits large textures into sections to avoid these limitations. This is usually used for large 640 by 480 or 800 by 600 background artwork.

`MenuScreen` has a profile property of `GuiContentProfile`, which is a standard Torque profile for large controls that will contain other controls. *Profiles* are collections of properties that can be applied in bulk to interface (or *gui*) objects. Profiles are much like style sheets (which you will be familiar with if you do any HTML programming), but using Torque Script syntax.

The definition of `GuiContentProfile` is pretty simple:

```
if(!IsObject(GuiContentProfile)) new GuiControlProfile (GuiContentProfile)
{
   opaque = true;
   fillColor = "255 255 255";
};
```

Basically, the object is opaque (no transparency allowed, even if an alpha channel exists in the object's source bitmap image). If the object doesn't fill the screen, then the unused screen space is filled with black (`RGB = 255 255 255`).

After the profile, the sizing and position information properties are set. See the sidebar titled "Profile Sizing Settings: horizSizing and vertSizing" for more information.

The `extent` property defines the horizontal and vertical dimensions of `MenuScreen`. The `minExtent` property specifies the smallest size that the object can have.

The `visible` property indicates whether the object can be scene on the screen. Using a "1" will make the object visible; a "0" will make it invisible.

The last significant property is the `bitmap` property—this specifies what bitmap image will be used for the background image of the object.

There are three `GuiButtonCtrl` objects contained in the `MenuScreen`. Most of the properties are the same as found in the `GuiChunkedBitmapCtrl`. But there are a few that are different and important.

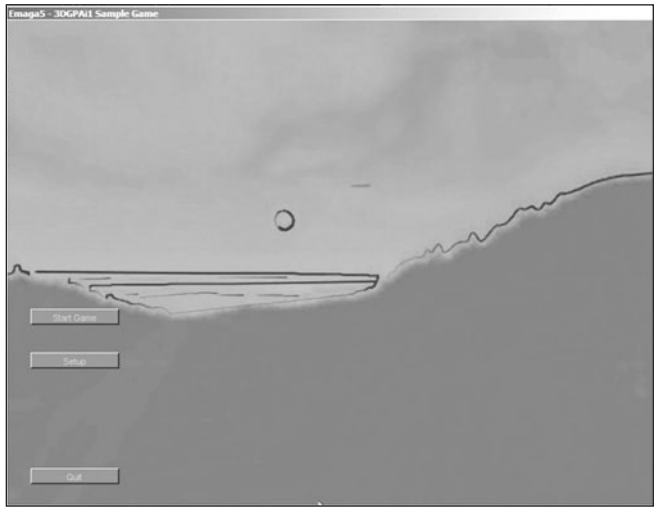


Figure 5.3 The Emaga5 MenuScreen.

The first is the `command` property. When the user clicks this button control, the function specified in the `command` property is executed.

Next, the `text` property is where you can enter the text label that will appear on the button.

Finally, the `buttonType` property is how you specify the particular visual style of the button.

Figure 5.3 shows the `MenuScreen` in all its glory.

Profile Sizing Settings: horizSizing and vertSizing

These settings are used to define how to resize or reposition an object when the object's container is resized. The outermost container is the `Canvas`; it will have a starting size of 640 pixels by 480 pixels. The `Canvas` and all of the objects within it will be resized or repositioned from this initial size.

When you resize a container, all of its child objects are resized and repositioned according to their `horizSizing` and `vertSizing` properties. The resizing action will be applied in a cascading manner to all subobjects in the object hierarchy.

The following property values are available:

Center	The object is positioned in the center of its container.
Relative	The object is resized and repositioned to maintain the same size and position relative to its container. If the parent size doubles, the object's size doubles as well.
Left	When the container is resized or moved, the change is applied to the distance between the object and the left edge of the screen.
Right	When the container is resized or moved, the change is applied to the distance between the object and the right edge of the screen.
Top	When the container is resized or moved, the change is applied to the distance between the object and the top edge of the screen.
Bottom	When the container is resized or moved, the change is applied to the distance between the object and the bottom edge of the screen.
Width	When the container is resized or moved, the change is applied to the extents of the object.
Height	When the container is resized or moved, the change is applied to the extents of the object itself.

control/client/interfaces/playerinterface.gui

The `PlayerInterface` control is the interface that is used during the game to display information in real time. The `Canvas` is the container for `PlayerInterface`. Type in the following code and save it as C:\Emaga5\control\client\interfaces\playerinterface.gui.

```
new GameTSCtrl(PlayerInterface) {
   profile = "GuiContentProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "0 0";
   extent = "640 480";
```

```
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
      noCursor = "1";
   new GuiCrossHairHud() {
      profile = "GuiDefaultProfile";
      horizSizing = "center";
      vertSizing = "center";
      position = "304 224";
      extent = "32 32";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      bitmap = "./interfaces/emaga_gunsight";
      wrap = "0";
      damageFillColor = "0.000000 1.000000 0.000000 1.000000";
      damageFrameColor = "1.000000 0.600000 0.000000 1.000000";
      damageRect = "50 4";
      damageOffset = "0 10";
   };
   new GuiHealthBarHud() {
      profile = "GuiDefaultProfile";
      horizSizing = "right";
      vertSizing = "top";
      position = "14 315";
      extent = "26 138";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      showFill = "1";
      displayEnergy = "0";
      showFrame = "1";
      fillColor = "0.000000 0.000000 0.000000 0.500000";
      frameColor = "0.000000 1.000000 0.000000 0.000000";
      damageFillColor = "0.800000 0.000000 0.000000 1.000000";
      pulseRate = "1000";
      pulseThreshold = "0.5";
         value = "1";
   };
   new GuiBitmapCtrl() {
      profile = "GuiDefaultProfile";
      horizSizing = "right";
```

```
   vertSizing = "top";
   position = "11 299";
   extent = "32 172";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   bitmap = "./interfaces/emaga_healthwidget";
   wrap = "0";
};
new GuiHealthBarHud() {
   profile = "GuiDefaultProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "53 315";
   extent = "26 138";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   showFill = "1";
   displayEnergy = "1";
   showFrame = "1";
   fillColor = "0.000000 0.000000 0.000000 0.500000";
   frameColor = "0.000000 1.000000 0.000000 0.000000";
   damageFillColor = "0.000000 0.000000 0.800000 1.000000";
   pulseRate = "1000";
   pulseThreshold = "0.5";
      value = "1";
};
new GuiBitmapCtrl() {
   profile = "GuiDefaultProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "50 299";
   extent = "32 172";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   bitmap = "./interfaces/emaga_healthwidget";
   wrap = "0";
};
new GuiTextCtrl(scorelabel) {
   profile = "ScoreTextProfile";
   horizSizing = "right";
```

```
      vertSizing = "bottom";
      position = "10 3";
      extent = "50 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "Score";
      maxLength = "255";
   };
   new GuiTextCtrl(Scorebox) {
      profile = "ScoreTextProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "50 3";
      extent = "100 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "0";
      maxLength = "255";
   };
};
```

`PlayerInterface` is the main `TSControl` through which the game is viewed; it also contains the HUD controls.

The object `GuiCrossHairHud` is the targeting crosshair. Use this to aim your weapons.

There are two `GuiHealthBarHud` controls, one for health and one for energy. It is essentially a vertical bar that indicates the state of health or energy of the player. Each `GuiHealthBarHud` is paired with a `GuiBitmapCtrl`, which is a bitmap that can be used to modify the appearance of the health and energy displays by overlaying on the `GuiHealthBarHud`.

note

HUD is a TLA (Three Letter Acronym) that means Heads Up Display. The expression is adopted from the world of high-tech military aircraft. The HUD comprises information and graphics that are projected onto the canopy or a small screen at eye level in front of the pilot. This allows the pilot to continue to look outside for threats, while still having instant visual access to flight- or mission-critical information. In game graphics the term HUD is used for visual displays that appear in-game, in a fashion that mirrors the real-world application.

There are two `GuiTextCtrl objects`, one for holding the accumulated score (`scorebox`) and one to provide a simple label for the scores box (`scorelabel`). We will be modifying the value of the `text` property from within the control source code in another module.

Team LRN

control/client/interfaces/splashscreen.gui

The `SplashScreen` control displays an informational screen (you saw it in Figure 5.2) when the game is started from Windows. A mouse click or key press makes this screen go away. Type in the following code and save it as C:\Emaga5\control\client\interfaces\splashscreen.gui.

```
new GuiChunkedBitmapCtrl(SplashScreen) {
   profile = "GuiDefaultProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   bitmap = "./interfaces/emaga_splash";
   useVariable = "0";
   tile = "0";
   noCursor=1;
   new GuiInputCtrl(SplashScreenInputCtrl) {
      profile = "GuiInputCtrlProfile";
      position = "0 0";
      extent = "10 10";
   };
};
```

The only thing special about this module is the new control `GuiInputCtrl`. This control is used to accept input from the user: mouse clicks, key presses, and so on. With this control defined we can then define our own handler methods for the control's object and therefore act upon the inputs. In our case here `SplashScreenInputCtrl::onInputEvent` is the handler method we've defined; it's contained in the client module we talked about earlier.

control/client/misc/screens.cs

The screen.cs module is where our programmed control and management activity is located. Type in the following code and save it as C:\Emaga5\control\client\misc\screens.cs.

```
//============================================================================
// control/client/misc/screens.cs
//
// Copyright (c) 2003 Kenneth C. Finney
//============================================================================
```

```
function PlayerInterface::onWake(%this)
{
   $enableDirectInput = "1";
   activateDirectInput();
   // just update the key map here
   playerKeymap.push();
}
function PlayerInterface::onSleep(%this)
{
   playerKeymap.pop();
}
function refreshBottomTextCtrl()
{
   BottomPrintText.position = "0 0";
}
function refreshCenterTextCtrl()
{
   CenterPrintText.position = "0 0";
}
function LoadScreen::onAdd(%this)
{
   %this.qLineCount = 0;
}
function LoadScreen::onWake(%this)
{
   CloseMessagePopup();
}
function LoadScreen::onSleep(%this)
{
   // Clear the load info:
   if ( %this.qLineCount !$= "" )
   {
      for ( %line = 0; %line < %this.qLineCount; %line++ )
         %this.qLine[%line] = "";
   }
   %this.qLineCount = 0;
   LOAD_MapName.setText( "" );
   LOAD_MapDescription.setText( "" );
   LoadingProgress.setValue( 0 );
   LoadingProgressTxt.setValue( "WAITING FOR SERVER" );
}
```

The methods in this module are representative of the sort of methods you can use for interface controls. You will probably use `OnWake` and `OnSleep` quite a bit in your interface scripts.

`OnWake` methods are called when an interface object is told to display itself, either by the `Canvas`'s `SetContent` or `PushDialog` methods.

`OnSleep` methods are called whenever an interface object is removed from display via the `PopDialog` method or when the `SetContent` call specifies a different object.

When `PushDialog` is used the interface that is shown operates like a modal dialog control—all input events are relayed through the dialog.

There is another pair of interface display methods for other objects called just `Push` and `Pop`. These will display the interface in a modeless manner, so that other controls or objects on the screen will still receive input events they are interested in.

`PlayerInterface::onWake` enables capturing mouse and keyboard inputs using `DirectInput`. It then makes the `PlayerKeymap` key bindings active using the `Push` method. When the `PlayerInterface` is removed from display, its `OnSleep` method removes the `PlayerKeymap` key bindings from consideration. You will need to ensure that you have defined global bindings for the user to employ; these will take over when the `PlayerKeymap` isn't in use anymore.

`RefreshBottomTextCtrl` and `RefreshCenterTextCtrl` just reposition these output controls to their default location on the screen, in case you have moved them somewhere else during the festivities.

`Loadscreen::OnWake` is called when we want to display the mission loading progress. It closes the message interface, if it happens to be open. The `Loadscreen` contents are modified elsewhere for us in the mission loading process, which is covered in Chapter 6.

When `Loadscreen::OnSleep` is called, it clears all of its text buffers and then outputs a message to indicate that all we need now is for the server to chime in.

control/client/misc/presetkeys.cs

Key bindings are the mapping of keyboard keys and mouse buttons to specific functions and commands. In a fully featured game we would provide the user with the ability to modify the key bindings using a graphical interface. Right now we will satisfy ourselves with creating a set of key bindings for the user, which we can keep around to be used as the initial defaults as we later expand our program.

Type in the following code and save it as C:\Emaga5\control\client\misc\presetkeys.cs.

```
//==============================================================================
// control/client/misc/presetkeys.cs
// Copyright (c) 2003 Kenneth C. Finney
//==============================================================================
```

```
if ( IsObject(PlayerKeymap) )  // If we already have a player key map,
   PlayerKeymap.delete();         // delete it so that we can make a new one
new ActionMap(PlayerKeymap);

function DoExitGame()
{
   MessageBoxYesNo( "Quit Mission", "Exit from this Mission?", "Quit();", "");
}
//============================================================================
// Motion Functions
//============================================================================
function GoLeft(%val)
{
   $mvLeftAction = %val;
}
function GoRight(%val)
{
   $mvRightAction = %val;
}
function GoAhead(%val)
{
   $mvForwardAction = %val;
}
function BackUp(%val)
{
   $mvBackwardAction = %val;
}
function DoYaw(%val)
{
   $mvYaw += %val * ($cameraFov / 90) * 0.02;
}
function DoPitch(%val)
{
   $mvPitch += %val * ($cameraFov / 90) * 0.02;
}
function DoJump(%val)
{
   $mvTriggerCount2++;
}
//============================================================================
// View Functions
//============================================================================
```

```
function Toggle3rdPPOVLook( %val )
{
   if ( %val )       $mvFreeLook = true;
   else             $mvFreeLook = false;
}
function MouseAction(%val)
{
   $mvTriggerCount0++;
}
function Toggle1stPPOV(%val)
{
   if (%val)
      $firstPerson = !$firstPerson;
}
function dropCameraAtPlayer(%val)
{
   if (%val)
      commandToServer('dropCameraAtPlayer');
}
function dropPlayerAtCamera(%val)
{
   if (%val)
      commandToServer('DropPlayerAtCamera');
}
function toggleCamera(%val)
{
   if (%val)
      commandToServer('ToggleCamera');
}
//============================================================================
// keyboard control mappings
//============================================================================
// available when player is in game
PlayerKeymap.Bind(mouse, button0, MouseAction ); // left mouse button
PlayerKeymap.Bind(keyboard, up, GoAhead);
PlayerKeymap.Bind(keyboard, down, BackUp);
PlayerKeymap.Bind(keyboard, left, GoLeft);
PlayerKeymap.Bind(keyboard, right, GoRight);
PlayerKeymap.Bind(keyboard, numpad0, DoJump );
PlayerKeymap.Bind(keyboard, z, Toggle3rdPPOVLook );
PlayerKeymap.Bind(keyboard, tab, Toggle1stPPOV );
PlayerKeymap.Bind(mouse, xaxis, DoYaw );
```

```
PlayerKeymap.Bind(mouse, yaxis, DoPitch );
PlayerKeymap.bind(keyboard, F8, dropCameraAtPlayer);
PlayerKeymap.bind(keyboard, F7, dropPlayerAtCamera);
PlayerKeymap.bind(keyboard, F6, toggleCamera);
// always available
GlobalActionMap.Bind(keyboard, escape, DoExitGame);
GlobalActionMap.Bind(keyboard, tilde, ToggleConsole);
```

The first three statements in this module prepare the `ActionMap`object, which we call `PlayerKeymap`. This is the set of key bindings that will prevail while we are actually in the game. Because this module is used in initial setup, we assume that there should not already be a `PlayerKeymapActionMap`, so we check to see if `PlayerKeymap` is an existing object, and if it is we delete it and create a new version.

We define a function to be called when we exit the game. It throws a `MessageBoxYesNo` dialog up on the screen, with the dialog's title set to the contents of the first parameter string. The second parameter string sets the contents of the dialog's prompt. The third parameter specifies the function to execute when the user clicks on the Yes button. The second parameter indicates what action to perform if the user clicks No—in this case nothing.

There are two other canned `MessageDialog` objects defined in the common code base: `MessageBoxOk`, which has no fourth parameter, and `MessageBoxOkCancel`, which accepts the same parameter set as `MessageBoxYesNo`.

Next we have a series of motion function definitions. Table 5.1 provides a description of the basic functions. These functions employ player event control triggers to do their dirty work. These triggers are described in detail in Chapter 6.

Of particular note in these functions is that they all have a single parameter, usually called `%val`. When functions are bound to keys or mouse buttons via a `Bind` method, the parameter is set to a nonzero value when the key or button is pressed and to 0 when the button is released. This allows us to create toggling functions, such as with `Toggle3rdPPOVLook`, which will be active only while the bound key is actually pressed.

After all the function definitions, we have the actual key bindings. With the `Bind` method, the first parameter is the input type, the second is the key or button identifier, and the third is the name of the function to be called.

After all the `PlayerKeymap` bindings, there are a few for `GlobalActionMap`, which is a globally predefined action map that is always available but can be overridden by other action maps. In this case we use `GlobalActionMap` for those bindings we want to be universally available.

Table 5.1 Basic Movement Functions

Command	Description
GoLeft and GoRight	Strafing to the left or the right.
GoAhead and BackUp	Running forward and backward.
DoYaw	Spinning or aiming horizontally by mouse or joystick control.
DoPitch	Looking vertically by mouse or joystick control.
DoJump	Momentary upward movement, with character animation.
Toggle3rdPPOVLook	Enables the "free look" feature. As long as the mapped key is pressed while the player is in third person view, the player can view his avatar by moving the mouse around.

Server Control Modules

Any game play features you want to implement should probably be done as a server control module, or part of one. If you are going to make a multiplayer online game, that *should* back there will change to a *must*. The only way we can ensure a level playing field and game play code security is to run the code on the server, and not on the client.

control/server/server.cs

On the server side, the server module is probably the single most influential module. It carries the server control oriented `GameConnection` methods for handling players and other game objects, as well as straightforward server control routines.

Type in the following code and save it as C:\Emaga5\control\server\server.cs.

```
//============================================================================
// control/server/server.cs
//  Copyright (c) 2003 by Kenneth C.  Finney.
//============================================================================
function OnServerCreated()
//----------------------------------------------------------------------------
// Once the engine has fired up the server, this function is called
//----------------------------------------------------------------------------
{
   Exec("./misc/camera.cs");
   Exec("./misc/shapeBase.cs");
   Exec("./misc/item.cs");
   Exec("./players/player.cs");
   Exec("./players/beast.cs");
   Exec("./players/ai.cs");
   Exec("./weapons/weapon.cs");
```

```
   Exec("./weapons/crossbow.cs");
}
function StartGame()
{
  if ($Game::Duration) // Start the game timer
    $Game::Schedule = Schedule($Game::Duration * 1000, 0, "onGameDurationEnd");
  $Game::Running = true;
  schedule( 2000, 0, "CreateBots");
}
function OnMissionLoaded()
{
   StartGame();
}
function OnMissionEnded()
{
   Cancel($Game::Schedule);
   $Game::Running = false;
}
function GameConnection::OnClientEnterGame(%this)
{
   // Create a new camera object.
   %this.camera = new Camera() {
      dataBlock = Observer;
   };
   MissionCleanup.Add( %this.camera );
   %this.SpawnPlayer();
}
function GameConnection::SpawnPlayer(%this)
{

   %this.CreatePlayer("0 0 201 1 0 0 0");
}
function GameConnection::CreatePlayer(%this, %spawnPoint)
{
   if (%this.player > 0)//The player should NOT already have an avatar object.
   {                       // If he does, that's a Bad Thing.
      Error( "Attempting to create an angus ghost!" );
   }
   // Create the player object
   %player = new Player() {
      dataBlock = HumanMaleAvatar;   // defined in players/player.cs
```

```
      client = %this;            // the avatar will have a pointer to its
   };                             // owner's GameConnection object
   %player.SetTransform(%spawnPoint); // where to put it
   // Update the camera to start with the player
   %this.camera.SetTransform(%player.GetEyeTransform());
   %player.SetEnergyLevel(100);
   // Give the client control of the player
   %this.player = %player;
   %this.setControlObject(%player);
}
function GameConnection::OnDeath(%this, %sourceObject, %sourceClient, %damageType,
%damLoc)
{
   // Switch the client over to the death cam and unhook the player object.
   if (IsObject(%this.camera) && IsObject(%this.player))
   {
      %this.camera.SetMode("Death",%this.player);
      %this.setControlObject(%this.camera);
   }
   %this.player = 0;
   if (%damageType $= "Suicide" || %sourceClient == %this)
   {
   }
   else
   {
     // good hit
   }
}
//============================================================================
// Server commands
//============================================================================
function ServerCmdToggleCamera(%client)
{
   %co = %client.getControlObject();
   if (%co == %client.player)
   {
      %co = %client.camera;
      %co.mode = toggleCameraFly;
   }
   else
   {
      %co = %client.player;
```

```
      %co.mode = observerFly;
   }
   %client.SetControlObject(%co);
}
function ServerCmdDropPlayerAtCamera(%client)
{
   if ($Server::DevMode || IsObject(EditorGui))
   {
      %client.player.SetTransform(%client.camera.GetTransform());
      %client.player.SetVelocity("0 0 0");
      %client.SetControlObject(%client.player);
   }
}
function ServerCmdDropCameraAtPlayer(%client)
{
   %client.camera.SetTransform(%client.player.GetEyeTransform());
   %client.camera.SetVelocity("0 0 0");
   %client.SetControlObject(%client.camera);
}
function ServerCmdUse(%client,%data)
{
   %client.GetControlObject().use(%data);
}
// stubs
function ClearCenterPrintAll()
{
}
function ClearBottomPrintAll()
{
}
```

The first function in this module, `OnServerCreated`, is pretty straightforward. When called, it loads all the specific game play modules we need.

After that comes `StartGame`, which is where we put stuff that is needed every time a new game starts. In this case if we have prescribed game duration, then we start the game timer using the `Schedule` function.

`Schedule` is an extremely important function, so we'll spend a little bit of time on it here. The usage syntax is:

```
%event = Schedule(time, reference, command, <param1...paramN>)
```

The function will schedule an event that will trigger in `time` milliseconds and execute `command` with parameters. If `reference` is not 0, then you need to make sure that `reference` is set

to be a valid object handle. When the reference object is deleted, the scheduled event is discarded if it hasn't already fired. The `Schedule` function returns an event ID number that can be used to track the scheduled event or cancel it later before it takes place.

In the case of our game timer, there is no game duration defined, so the game is open-ended, and the `Schedule` call will not take place. If, for example, `$Game::Duration` had been set to 1,800 (for 30 minutes times 60 seconds per minute), then the call to schedule would have had the first parameter set to 1,800 times 1,000, or 1,800,000, which is the number of milliseconds in 30 minutes.

`OnMissionLoaded` is called by `LoadMission` once the mission is finished loading. All it really does is start up the game play, but this is an ideal location to insert code that needs to adjust its capabilities based upon whatever mission was loaded.

The next function, `OnMissionEnded`, is called at the conclusion of the running of a mission, usually in the `DestroyServer` function. Here it cancels the end-of-game event that has been scheduled; if no game duration was scheduled—as is our case at the moment—then nothing happens, quietly.

After that is the `GameConnection::OnClientEnterGame` method. This method is called when the client has been accepted into the game by the server—the client has not actually entered the game yet though. The server creates a new observer mode camera and adds it to the `MissionCleanup` group. This group is used to contain objects that will need to be removed from memory when a mission is finished. Then we initiate the spawning of the player's avatar into the game world.

The `GameConnection::SpawnPlayer` is a "glue" method, which will have more functionality in the future. Right now we use it to call the `CreatePlayer` method with a fixed transform to tell it where to place the newly created player-avatar. Normally this is where we would place the player spawn decision code. It might also call a function that would figure out the spawn point transforms by looking up spawn markers. Once we know where the player will spawn, then we would create the avatar by calling `CreatePlayer`.

`GameConnection::CreatePlayer` is the method that creates the player's avatar object, sets it up, and then passes control of the avatar to the player. The first thing to watch out for is that we must ensure that the `GameConnection` does not already, or still, have an avatar assigned to it. If it does, then we risk creating what the GarageGames guys call an Angus Ghost. This is a ghosted object, on all the clients, that has no controlling client scoped to it. We don't want that! Once that is sorted out, we create the new avatar, give it some energy, and pass control to the player, the same way we did previously in Chapter 4.

Team LRN

`GameConnection::onDeath` is called from a player's `Damage` handler method if the player's damage exceeds a certain amount. What we do is switch the client over to the death cam and unhook the player object. This allows the player to swivel his view in orbit around the "corpse" of his avatar until he decides to respawn. There is a code block containing the comment "good hit" where we would add code to provide points scoring and other game play functionality if we want it. We can also penalize a player for committing suicide, by either evaluating the damage type or the ID of the owner of the weapon that killed the player.

There then are a series of `ServerCmd` message handlers that change whether the player controls the camera or the avatar based on the message received.

`ServerCmdToggleCamera` alternates between attaching the player to the camera or to his avatar as the control object. Each time the function is called, it checks to see which object is the control object—camera or avatar—and then selects the other one to be the new control object.

`ServerCmdDropPlayerAtCamera` will move the player's avatar to wherever the camera object is currently located and sets the player-avatar's velocity to 0. The control object is always set to be the player's avatar when the function exits.

`ServerCmdDropCameraAtPlayer` does just the opposite. It sets the camera's transform to match the player-avatar's and then sets the velocity to 0. The control object is always set to be the camera when the function exits.

The next function, `ServerCmdUse`, is an important game play message handler. We call this function whenever we want to activate or otherwise use an object that the player controls, "has mounted," or holds in inventory. When called, this function figures out the handle of the client's control object and then passes the data it has received to that object's use method. The data can be anything but is often the activation mode or sometimes a quantity (like a powerup or health value). You'll see how the back end of this works later in the item module.

control/server/players/player.cs

This is "the biggie." You will probably spend more time working with, tweaking, adjusting, and yes, possibly even cursing this module—or your own variations of this module—than any other.

Type in the following code and save it as C:\Emaga5\control\server\players\player.cs.

```
//--------------------------------------------------------------------------
// control/server/players/player.cs
//  Copyright (c) 2003 by Kenneth C.  Finney.
//--------------------------------------------------------------------------
datablock PlayerData(HumanMaleAvatar)
```

```
{
   className = MaleAvatar;
   shapeFile = "~/data/models/avatars/human/player.dts";
   emap = true;
   renderFirstPerson = false;
   cameraMaxDist = 3;
   mass = 100;
   density = 10;
   drag = 0.1;
   maxdrag = 0.5;
   maxDamage = 100;
   maxEnergy =  100;
   maxForwardSpeed = 15;
   maxBackwardSpeed = 10;
   maxSideSpeed = 12;
   minJumpSpeed = 20;
   maxJumpSpeed = 30;
   runForce = 1000;
   jumpForce = 1000;
   runSurfaceAngle  = 40;
   jumpSurfaceAngle = 30;
   runEnergyDrain = 0.05;
   minRunEnergy = 1;
   jumpEnergyDrain = 20;
   minJumpEnergy = 20;
   recoverDelay = 30;
   recoverRunForceScale = 1.2;
   minImpactSpeed = 10;
   speedDamageScale = 3.0;
   repairRate = 0.03;
   maxInv[Copper] = 9999;
   maxInv[Silver] = 99;
   maxInv[Gold] = 9;
   maxInv[Crossbow] = 1;
   maxInv[CrossbowAmmo] = 20;
};
//============================================================================
// Avatar Datablock methods
//============================================================================
function MaleAvatar::onAdd(%this,%obj)
{
   %obj.mountVehicle = false;
```

```
   // Default dynamic Avatar stats
   %obj.setRechargeRate(0);
   %obj.setRepairRate(%this.repairRate);
}
function MaleAvatar::onRemove(%this, %obj)
{
   %client = %obj.client;
   if (%client.player == %obj)
   {
      %client.player = 0;
   }
}
function MaleAvatar::onCollision(%this,%obj,%col,%vec,%speed)
{
  %obj_state = %obj.getState();
  %col_className = %col.getClassName();
  %col_dblock_className = %col.getDataBlock().className;
  %colName = %col.getDataBlock().getName();
  if ( %obj_state $= "Dead")
    return;
  if (%col_className $= "Item" || %col_className $= "Weapon" )
  {
    %obj.pickup(%col);
  }
}
//----------------------------------------------------------------------------
// HumanMaleAvatar (ShapeBase) class methods
//----------------------------------------------------------------------------
function HumanMaleAvatar::onImpact(%this,%obj,%collidedObject,%vec,%vecLen)
{
   %obj.Damage(0, VectorAdd(%obj.getPosition(),%vec),
      %vecLen * %this.speedDamageScale, "Impact");
}
function HumanMaleAvatar::Damage(%this, %obj, %sourceObject, %position, %damage,
%damageType)
{
   if (%obj.getState() $= "Dead")
      return;
   %obj.applyDamage(%damage);
   %location = "Body";
   %client = %obj.client;
   %sourceClient = %sourceObject ? %sourceObject.client : 0;
```

```
   if (%obj.getState() $= "Dead")
   {
      %client.onDeath(%sourceObject, %sourceClient, %damageType, %location);
   }
}
function HumanMaleAvatar::onDamage(%this, %obj, %delta)
{
   if (%delta > 0 && %obj.getState() !$= "Dead")
   {
      // Increment the flash based on the amount.
      %flash = %obj.getDamageFlash() + ((%delta / %this.maxDamage) * 2);
      if (%flash > 0.75)
         %flash = 0.75;

      if (%flash > 0.001)
      {
        %obj.setDamageFlash(%flash);
      }
      %obj.setRechargeRate(-0.0005);
      %obj.setRepairRate(0.0005);
   }
}
function HumanMaleAvatar::onDisabled(%this,%obj,%state)
{
   %obj.clearDamageDt();
   %obj.setRechargeRate(0);
   %obj.setRepairRate(0);
   %obj.setImageTrigger(0,false);
   %obj.schedule(5000, "startFade", 5000, 0, true);
   %obj.schedule(10000, "delete");
}
```

The first code block is a data block definition for a data block called HumanMaleAvatar of the PlayerData data block class. Table 5.2 provides a quick reference description of the items in this data block.

Table 5.2 Emaga5 Avatar Properties

Property	DescriptionNeural Network
`className`	Defines an arbitrary class that the avatar can belong to.
`shapeFile`	Specifies the file that contains the 3D model of the avatar.
`emap`	Enables environment mapping on the avatar model.
`renderFirstPerson`	When true, causes the avatar model to be visible when in first-person point-of-view mode.
`cameraMaxDist`	Maximum distance from the avatar for the camera in third-person point-of-view mode.
`mass`	The mass of the avatar in terms of the game world.
`density`	Arbitrarily defines density. Low-density players will float in water.
`drag`	Slows down the avatar through simulated friction.
`maxDrag`	Maximum allowable drag.
`maxEnergy`	Maximum energy allowed.
`maxDamage`	Maximum damage points that can be sustained before avatar is killed.
`maxForwardSpeed`	Maximum speed allowable when moving forward.
`maxBackwardSpeed`	Maximum speed allowable when moving backward.
`maxSideSpeed`	Maximum speed allowable when moving sideways (strafing).
`minJumpSpeed`	Below this speed, you can't make the avatar jump.
`maxJumpSpeed`	Above this speed, you can't make the avatar jump.
`jumpForce`	The force, and therefore the acceleration, when jumping.
`runForce`	The force, and therefore the acceleration, when starting to run.
`runSurfaceAngle`	Maximum slope (in degrees) that the avatar can run on.
`jumpSurfaceAngle`	Maximum slope (in degrees) that the avatar can jump on, usually somewhat less than `RunSurfaceAngle`.
`runEnergyDrain`	How quickly energy is lost when the player is running.
`minRunEnergy`	Below this, the player will not move.
`jumpEnergyDrain`	How quickly energy is lost when the player jumps.
`minJumpEnergy`	Below this, the player can't jump anymore.
`recoverDelay`	How long it takes after a landing from a fall or jump, measured in ticks, where 1 tick = 32 milliseconds.
`recoverRunForceScale`	How much to scale the run force by while in the postlanding recovery state.
`minImpactSpeed`	Above this speed, an impact will cause damage.
`speedDamageScale`	Used to impart speed-scaled damage.
`repairRate`	How quickly damage is repaired when first aid or health is applied.
`maxInv[Copper]`	Maximum number of copper coins that the player can carry.
`maxInv[Silver]`	Maximum number of silver coins that the player can carry.
`maxInv[Gold]`	Maximum number of gold coins that the player can carry.
`maxInv[Crossbow]`	Maximum number of crossbows that the player can carry.
`maxInv[CrossbowAmmo]`	Maximum amount of crossbow ammunition that the player can carry.

Team LRN

A brief word about the `classname` property: It's a GameBase classname property for this data block, which in this case is `MaleAvatar`. We use this class name to provide a place to hang various methods, which are defined later in the module.

In Chapter 3 we encountered *environment mapping*, which is a rendering technique that provides a method of taking the game world appearance and surroundings into account when rendering an object. You can enable environment mapping when rendering the avatar model by setting the `emap` property to `true`.

If we set the property `renderFirstPerson` to `true`, then when we are playing in first-person point-of-view mode, we will be able to see our avatar, our "body," as we look around. With it set to `false`, then we won't see it, no matter which way we look.

To control your avatar's energy depletion, you will want to adjust the following properties: `maxEnergy`, `runEnergyDrain`, `minRunEnergy`, `jumpEnergyDrain`, and `minJumpEnergy`. Generally, the minimum jump energy should be set higher than the minimum run energy. Also, jump energy drain should be faster, thus a higher number, than the run energy drain value.

Next is a series of methods that are used when dealing with the avatar as a GameBase class.

The first, the `MaleAvatar::onAdd`, is the method called whenever a new instance of an avatar is added to the game. In this case we initialize a few variables and then transfer the value of the data block's `repairRate` property (remember that a data block is static and unchangeable once transferred to the client) to `Player` object in order to have it available for later use. The `%obj` parameter refers to the `Player` object handle.

Of course, we also need to know what to do when it's time to remove the avatar, which is what `MaleAvatar::onRemove` does. It's nothing spectacular—it just sets the handle properties to 0 and moves on.

One of the methods that gets the most exercise from a healthy and active avatar is the `MaleAvatar::onCollision` method. This method is called by the engine whenever it establishes that the avatar has collided with some other collision-enabled object. Five parameters are provided: The first is the handle of this data block, the second is the handle of the player object, the third is the handle of the object that hit us (or that we hit), the fourth is the relative velocity vector between us and the object we hit, and the fifth is the scalar speed of the object we hit. Using these inputs, we can do some pretty fancy collision calculations.

What we do, though, is just find out what the state of our avatar is (alive or dead) and what kind of object we hit. If we are dead (our avatar's body could be sliding down a hill, for example), we bail out of this method; otherwise we try to pick up the item we hit, providing it is an item or a weapon.

The engine calls `HumanMaleAvatar::onImpact` when our avatar hits something. Unlike `onCollision`, this method detects *any* sort of impact, not just a collision with an item in the world. Collisions occur between ShapeBase class things, like items, player avatars, vehicles,

and weapons. Impacts occur with those things, as well as terrain and interiors. So, `onImpact` provides essentially the same five parameters. We use that data to calculate how much damage the player should incur, and we apply that damage to the avatar's object using its `Damage` method.

The `HumanMaleAvatar::Damage` is where we try to ascertain what effect on the avatar the damage will have. If we want to implement hit boxes, or damage calculations based on object components, we would do that here. In this case if the player is dead, we again bail. If not, we apply the damage (which increases the accumulated damage value) and then obtain the object's current state. If the object is now dead, we call the `OnDeath` handler and exit the function.

Next is the `HumanMaleAvatar::onDamage` method, which is activated by the engine whenever the object's damage value changes. This is the method we want to use when applying some sort of special effect to the player when damage occurs—like making the screen flash or using some audio. In this case we do flash the screen, and we also start a slow energy drain caused by the damage. At the same time, we start a slow damage repair, which means that after some period of time, we will have regained some of our health (negative health equals positive damage).

When the player's damage exceeds the `maxDamage` value, the player object is set to the *disabled* state. When that happens, the function `HumanMaleAvatar::onDisabled` is called. This is where we deal with the final stages of the death of a player's avatar. What we are doing is resetting all the various repair values, disabling any mounted weapons, and then beginning the process of disposing of the corpse. We keep it around for a few seconds before letting it slowly fade away.

control/server/weapons/weapon.cs

The tutorial install kit doesn't like to create empty folders, so you will have to create the weapons folder in the server tree, as follows: C:\Emaga5\control\server\weapons\.

This Weapon module contains *name space* helper methods for Weapon and Ammo classes that define a set of methods that are part of dynamic name spaces class. All ShapeBase class images are mounted into one of eight slots on a shape.

There are also hooks into the inventory system specifically for use with weapons and ammo. Go ahead and type in the following module and save it as C:\Emaga5\control\server\weapons\weapon.cs.

```
//--------------------------------------------------------------------------
// control/server/weapons/weapon.cs
// Copyright (c) 2003 Kenneth C. Finney
// Portions Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
```

```
//============================================================================
$WeaponSlot = 0;
function Weapon::OnUse(%data,%obj)
{
   if (%obj.GetMountedImage($WeaponSlot) != %data.image.GetId())
   {
      %obj.mountImage(%data.image, $WeaponSlot);
      if (%obj.client)
         MessageClient(%obj.client, 'MsgWeaponUsed', '\cOWeapon selected');
   }
}
function Weapon::OnPickup(%this, %obj, %shape, %amount)
{
   if (Parent::OnPickup(%this, %obj, %shape, %amount))
   {
      if ( (%shape.GetClassName() $= "Player" ||
            %shape.GetClassName() $= "AIPlayer"  )  &&
            %shape.GetMountedImage($WeaponSlot) == 0)
      {
         %shape.Use(%this);
      }
   }
}
function Weapon::OnInventory(%this,%obj,%amount)
{
   if (!%amount && (%slot = %obj.GetMountSlot(%this.image)) != -1)
      %obj.UnmountImage(%slot);
}
function WeaponImage::OnMount(%this,%obj,%slot)
{
   if (%obj.GetInventory(%this.ammo))
      %obj.SetImageAmmo(%slot,true);
}
function Ammo::OnPickup(%this, %obj, %shape, %amount)
{
   if (Parent::OnPickup(%this, %obj, %shape, %amount))
   {

   }
}
function Ammo::OnInventory(%this,%obj,%amount)
{
```

```
   for (%i = 0; %i < 8; %i++)
   {
      if ((%image = %obj.GetMountedImage(%i)) > 0)
         if (IsObject(%image.ammo) && %image.ammo.GetId() == %this.GetId())
            %obj.SetImageAmmo(%i,%amount != 0);
   }
}
function RadiusDamage(%sourceObject, %position, %radius, %damage, %damageType, %impulse)
{
   InitContainerRadiusSearch(%position, %radius, $TypeMasks::ShapeBaseObjectType);

   %halfRadius = %radius / 2;
   while ((%targetObject = ContainerSearchNext()) != 0) {
      %coverage = CalcExplosionCoverage(%position, %targetObject,
         $TypeMasks::InteriorObjectType |  $TypeMasks::TerrainObjectType |
         $TypeMasks::ForceFieldObjectType | $TypeMasks::VehicleObjectType);
      if (%coverage == 0)
         continue;
      %dist = ContainerSearchCurrRadiusDist();
      %distScale = (%dist < %halfRadius)? 1.0:
         1.0 - ((%dist - %halfRadius) / %halfRadius);
      %targetObject.Damage(%sourceObject, %position,
         %damage * %coverage * %distScale, %damageType);
      if (%impulse) {
         %impulseVec = VectorSub(%targetObject.GetWorldBoxCenter(), %position);
         %impulseVec = VectorNormalize(%impulseVec);
         %impulseVec = VectorScale(%impulseVec, %impulse * %distScale);
         %targetObject.ApplyImpulse(%position, %impulseVec);
      }
   }
}
```

The weapon management system contained in this module assumes all primary weapons are mounted into the slot specified by the `$WeaponSlot` variable.

The first method defined, `Weapon::onUse`, describes the default behavior for all weapons when used: mount it into the object's `$WeaponSlot` weapon slot, which is currently set to slot 0. A message is sent to the client indicating that the mounting action was successful. Picture this: You are carrying a holstered pistol. When the Use command is sent to the server after being initiated by some key binding, the pistol is removed from the holster, figuratively speaking, and placed in image slot 0, where it becomes visible in the player's hand. That's what takes place when you "use" a weapon.

Team LRN

The next method, `Weapon::onPickup`, is the weapon's version of what happens when you collide with a weapon, and the `onCollision` method of the `MaleAvatar` decides you need to pick this weapon up. First, the parent `Item` method performs the actual pickup, which involves the act of including the weapon in our inventory. After that has been handled, we get control of the process here. What we do is automatically use the weapon if the player does not already have one in hand.

When the Item inventory code detects a change in the inventory status, the `Weapon::onInventory` method is called in order to check if we are holding an instance of the weapon in a mount slot, in case there are none showing in inventory. When the weapon inventory has changed, make sure there are no weapons of this type mounted if there are none left in inventory.

The method `WeaponImage::onMount` is called when a weapon is mounted (used). We use it to set the state according to the current inventory.

If there are any special effects we want to invoke when we pick up a weapon, we would put them in the `Ammo::onPickup` method. The parent `Item` method performs the actual pickup, and then we take a crack at it. If we had booby-trapped weapons, this would be a good place to put the code.

Generally, ammunition is treated as an item in its own right. The `Ammo::onInventory` method is called when ammo inventory levels change. Then we can update any mounted images using this ammo to reflect the new state. In the method we cycle through all the mounted weapons to examine each mounted weapon's ammo status.

`RadiusDamage` is a pretty nifty function that we use to apply explosion effects to objects within a certain distance from where the explosion occurred and to impart an impulse force on each object to move it if called for.

The first statement in the function uses `InitContainerRadiusSearch` to prepare the container system for use. It basically indicates that the engine is going to search for all objects of the type `$TypeMasks::ShapeBaseObjectType` located within `%radius` distance from the location specified by `%position`. See Table A.1 in Appendix A for a list of available type masks. Once the container radius search has been set up, we then will make successive calls to `ContainerSearchNext`. Each call will return the handle of the objects found that match the mask we supplied. If the handle is returned as 0, then the search has finished.

So we enter a nicely sized `while` loop that will continue as long as `ContainerSearchNext` returns a valid object handle (nonzero) in `%targetObject`. With each object found, we calculate how much of the object is affected by the explosion but only apply this calculation based on how much of the explosion is blocked by certain types of objects. If an object of one of these types has completely blocked the explosion, then the explosion coverage will be 0.

Team LRN

Then we use the `ContainerSearchCurrRadiusDist` to find the approximate radius of the affected object and subtract that value from the center-of-explosion to center-of-object distance to get the distance to the nearest surface of the object. Next, damage is applied that is proportional to this distance. If the nearest surface of the object is less than half the radius of the explosion away, then full damage is applied.

Finally, a proportional impulse force vector, if appropriate, is applied using modified distance scale. This has the effect of pushing the object away from the center of the blast.

control/server/weapons/crossbow.cs

For each weapon in our game, we need a definition module that contains the specifics for that weapon—its data blocks, methods, particle definitions (if they are going to be unique to the weapon), and other useful stuff.

There is a lot of material here, so if you want to exclude some stuff to cut back on typing, then leave out all of the particle and explosion data blocks. You won't get any cool-looking explosions or smoke trails, and you will get some error warnings in your console log file, but the weapon will still work.

The crossbow is a somewhat stylized and fantasy-based crossbow—rather medieval in flavor. It fires a burning bolt projectile that explodes like a grenade on impact. It's cool.

Type in the following code and save it as C:\Emaga5\control\server\weapons\crossbow.cs.

```
//============================================================================
// control/server/weapons/crossbow.cs
// Copyright (c) 2003 Kenneth C. Finney
// Portions Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//============================================================================
datablock ParticleData(CrossbowBoltParticle)
{
   textureName          = "~/data/particles/smoke";
   dragCoefficient     = 0.0;
   gravityCoefficient   = -0.2;   // rises slowly
   inheritedVelFactor   = 0.00;
   lifetimeMS           = 500;  // lasts 0.7 second
   lifetimeVarianceMS   = 150;   // ...more or less
   useInvAlpha = false;
   spinRandomMin = -30.0;
   spinRandomMax = 30.0;
   colors[0]     = "0.56 0.36 0.26 1.0";
   colors[1]     = "0.56 0.36 0.26 1.0";
   colors[2]     = "0 0 0 0";
```

```
   sizes[0]      = 0.25;
   sizes[1]      = 0.5;
   sizes[2]      = 1.0;
   times[0]      = 0.0;
   times[1]      = 0.3;
   times[2]      = 1.0;
};
datablock ParticleEmitterData(CrossbowBoltEmitter)
{
   ejectionPeriodMS = 10;
   periodVarianceMS = 5;
   ejectionVelocity = 0.25;
   velocityVariance = 0.10;
   thetaMin          = 0.0;
   thetaMax          = 90.0;
   particles = CrossbowBoltParticle;
};
datablock ParticleData(CrossbowExplosionParticle)
{
   textureName           = "~/data/particles/smoke";
   dragCoefficient      = 2;
   gravityCoefficient   = 0.2;
   inheritedVelFactor   = 0.2;
   constantAcceleration = 0.0;
   lifetimeMS            = 1000;
   lifetimeVarianceMS   = 150;
   colors[0]     = "0.56 0.36 0.26 1.0";
   colors[1]     = "0.56 0.36 0.26 0.0";
   sizes[0]      = 0.5;
   sizes[1]      = 1.0;
};
datablock ParticleEmitterData(CrossbowExplosionEmitter)
{
   ejectionPeriodMS = 7;
   periodVarianceMS = 0;
   ejectionVelocity = 2;
   velocityVariance = 1.0;
   ejectionOffset   = 0.0;
   thetaMin          = 0;
   thetaMax          = 60;
   phiReferenceVel  = 0;
   phiVariance      = 360;
```

```
   particles = "CrossbowExplosionParticle";
};
datablock ParticleData(CrossbowExplosionSmoke)
{
   textureName          = "~/data/particles/smoke";
   dragCoefficient     = 100.0;
   gravityCoefficient   = 0;
   inheritedVelFactor   = 0.25;
   constantAcceleration = -0.80;
   lifetimeMS           = 1200;
   lifetimeVarianceMS   = 300;
   useInvAlpha =  true;
   spinRandomMin = -80.0;
   spinRandomMax =  80.0;

   colors[0]     = "0.56 0.36 0.26 1.0";
   colors[1]     = "0.2 0.2 0.2 1.0";
   colors[2]     = "0.0 0.0 0.0 0.0";

   sizes[0]      = 1.0;
   sizes[1]      = 1.5;
   sizes[2]      = 2.0;

   times[0]      = 0.0;
   times[1]      = 0.5;
   times[2]      = 1.0;

};
datablock ParticleEmitterData(CrossbowExplosionSmokeEmitter)
{
   ejectionPeriodMS = 10;
   periodVarianceMS = 0;
   ejectionVelocity = 4;
   velocityVariance = 0.5;
   thetaMin         = 0.0;
   thetaMax         = 180.0;
   lifetimeMS       = 250;
   particles = "CrossbowExplosionSmoke";
};
datablock ParticleData(CrossbowExplosionSparks)
{
   textureName          = "~/data/particles/spark";
```

```
   dragCoefficient      = 1;
   gravityCoefficient   = 0.0;
   inheritedVelFactor   = 0.2;
   constantAcceleration = 0.0;
   lifetimeMS           = 500;
   lifetimeVarianceMS   = 350;
   colors[0]     = "0.60 0.40 0.30 1.0";
   colors[1]     = "0.60 0.40 0.30 1.0";
   colors[2]     = "1.0 0.40 0.30 0.0";
   sizes[0]      = 0.5;
   sizes[1]      = 0.25;
   sizes[2]      = 0.25;

   times[0]      = 0.0;
   times[1]      = 0.5;
   times[2]      = 1.0;
};
datablock ParticleEmitterData(CrossbowExplosionSparkEmitter)
{
   ejectionPeriodMS = 3;
   periodVarianceMS = 0;
   ejectionVelocity = 13;
   velocityVariance = 6.75;
   ejectionOffset   = 0.0;
   thetaMin         = 0;
   thetaMax         = 180;
   phiReferenceVel  = 0;
   phiVariance      = 360;
   overrideAdvances = false;
   orientParticles  = true;
   lifetimeMS       = 100;
   particles = "CrossbowExplosionSparks";
};
datablock ExplosionData(CrossbowSubExplosion1)
{
   offset = 1.0;
   emitter[0] = CrossbowExplosionSmokeEmitter;
   emitter[1] = CrossbowExplosionSparkEmitter;
};
datablock ExplosionData(CrossbowSubExplosion2)
{
   offset = 1.0;
```

```
   emitter[0] = CrossbowExplosionSmokeEmitter;
   emitter[1] = CrossbowExplosionSparkEmitter;
};
datablock ExplosionData(CrossbowExplosion)
{
   lifeTimeMS = 1200;
   particleEmitter = CrossbowExplosionEmitter; // Volume particles
   particleDensity = 80;
   particleRadius = 1;
   emitter[0] = CrossbowExplosionSmokeEmitter;   // Point emission
   emitter[1] = CrossbowExplosionSparkEmitter;
   subExplosion[0] = CrossbowSubExplosion1;  // Sub explosion objects
   subExplosion[1] = CrossbowSubExplosion2;
   shakeCamera = true;                  // Camera Shaking
   camShakeFreq = "10.0 11.0 10.0";
   camShakeAmp = "1.0 1.0 1.0";
   camShakeDuration = 0.5;
   camShakeRadius = 10.0;
   lightStartRadius = 6;                // Dynamic light
   lightEndRadius = 3;
   lightStartColor = "0.5 0.5 0";
   lightEndColor = "0 0 0";
};
datablock ProjectileData(CrossbowProjectile)
{
   projectileShapeName = "~/data/models/weapons/bolt.dts";
   directDamage        = 20;
   radiusDamage        = 20;
   damageRadius        = 1.5;
   explosion           = CrossbowExplosion;
   particleEmitter     = CrossbowBoltEmitter;
   muzzleVelocity      = 100;
   velInheritFactor    = 0.3;
   armingDelay         = 0;
   lifetime            = 5000;
   fadeDelay           = 5000;
   bounceElasticity    = 0;
   bounceFriction      = 0;
   isBallistic         = true;
   gravityMod = 0.80;
   hasLight    = true;
   lightRadius = 4.0;
```

```
   lightColor  = "0.5 0.5 0";
};
function CrossbowProjectile::OnCollision(%this,%obj,%col,%fade,%pos,%normal)
{
   if (%col.getType() & $TypeMasks::ShapeBaseObjectType)
      %col.damage(%obj,%pos,%this.directDamage,"CrossbowBolt");
RadiusDamage(%obj,%pos,%this.damageRadius,%this.radiusDamage,"CrossbowBolt",0);
}
datablock ItemData(CrossbowAmmo)
{
   category = "Ammo";
   className = "Ammo";
   shapeFile = "~/data/models/weapons/boltclip.dts";
   mass = 1;
   elasticity = 0.2;
   friction = 0.6;

       // Dynamic properties defined by the scripts
       pickUpName = "crossbow bolts";
   maxInventory = 20;
};
datablock ItemData(Crossbow)
{
   category = "Weapon";
   className = "Weapon";
   shapeFile = "~/data/models/weapons/crossbow.dts";
   mass = 1;
   elasticity = 0.2;
   friction = 0.6;
   emap = true;
   pickUpName = "a crossbow";
   image = CrossbowImage;
};
datablock ShapeBaseImageData(CrossbowImage)
{
   shapeFile = "~/data/models/weapons/crossbow.dts";
   emap = true;
   mountPoint = 0;
   eyeOffset = "0.1 0.4 -0.6";
   correctMuzzleVector = false;
   className = "WeaponImage";
   item = Crossbow;
```

```
   ammo = CrossbowAmmo;
   projectile = CrossbowProjectile;
   projectileType = Projectile;

   stateName[0]                       = "Preactivate";
   stateTransitionOnLoaded[0]        = "Activate";
   stateTransitionOnNoAmmo[0]        = "NoAmmo";
   stateName[1]                       = "Activate";
   stateTransitionOnTimeout[1]       = "Ready";
   stateTimeoutValue[1]               = 0.6;
   stateSequence[1]                   = "Activate";
   stateName[2]                       = "Ready";
   stateTransitionOnNoAmmo[2]        = "NoAmmo";
   stateTransitionOnTriggerDown[2]   = "Fire";
   stateName[3]                       = "Fire";
   stateTransitionOnTimeout[3]       = "Reload";
   stateTimeoutValue[3]               = 0.2;
   stateFire[3]                       = true;
   stateRecoil[3]                     = LightRecoil;
   stateAllowImageChange[3]          = false;
   stateSequence[3]                   = "Fire";
   stateScript[3]                     = "onFire";
   stateName[4]                       = "Reload";
   stateTransitionOnNoAmmo[4]        = "NoAmmo";
   stateTransitionOnTimeout[4]       = "Ready";
   stateTimeoutValue[4]               = 0.8;
   stateAllowImageChange[4]          = false;
   stateSequence[4]                   = "Reload";
   stateEjectShell[4]                 = true;
   stateName[5]                       = "NoAmmo";
   stateTransitionOnAmmo[5]          = "Reload";
   stateSequence[5]                   = "NoAmmo";
   stateTransitionOnTriggerDown[5]   = "DryFire";
   stateName[6]                       = "DryFire";
   stateTimeoutValue[6]               = 1.0;
   stateTransitionOnTimeout[6]       = "NoAmmo";
};
function CrossbowImage::onFire(%this, %obj, %slot)
{
   %projectile = %this.projectile;
   %obj.decInventory(%this.ammo,1);
   %muzzleVector = %obj.getMuzzleVector(%slot);
```

```
   %objectVelocity = %obj.getVelocity();
   %muzzleVelocity = VectorAdd(
      VectorScale(%muzzleVector, %projectile.muzzleVelocity),
      VectorScale(%objectVelocity, %projectile.velInheritFactor));
   %p = new (%this.projectileType)() {
      dataBlock        = %projectile;
      initialVelocity  = %muzzleVelocity;
      initialPosition  = %obj.getMuzzlePoint(%slot);
      sourceObject     = %obj;
      sourceSlot       = %slot;
      client           = %obj.client;
   };
   MissionCleanup.add(%p);
   return %p;
}
```

We will cover the contents of the particle, explosion, and weapon datablocks in detail in later chapters when we start creating our own weapons. Therefore we will skip discussion of these elements for now and focus on the data block's methods.

The first method, and one of the most critical, is the `CrossbowProjectile::OnCollision` method. When called, it looks first to see if the projectile has collided with the right kind of object. If so, then the projectile's damage value is applied directly to the struck object. The method then calls the `RadiusDamage` function to apply damage to surrounding objects, if applicable.

When shooting the crossbow, the `CrossbowImage::onFire` method is used to handle the aspects of firing the weapon that cause the projectile to be created and launched. First, the projectile is removed from inventory, and then a vector is calculated based upon which way the muzzle is facing. This vector is scaled by the specified muzzle velocity of the projectile and the velocity inherited from the movement of the crossbow (which gets that velocity from the movement of the player).

Finally, a new projectile object is spawned into the game world at the location of the weapon's muzzle—the projectile possesses all of the velocity information at the time of spawning, so when added, it immediately begins coursing toward its target.

The projectile is added to the `MissionCleanup` group before the method exits.

control/server/misc/item.cs

This module contains the code needed to pick up and create items, as well as definitions of specific items and their methods. Type in the following code and save it as C:\Emaga5\control\server\misc\item.cs.

```
//=============================================================================
// control/server/misc/item.cs
//  Copyright (c) 2003 by Kenneth C.  Finney.
//=============================================================================
$RespawnDelay = 20000;
$LoiterDelay = 10000;
function Item::Respawn(%this)
{
   %this.StartFade(0, 0, true);
   %this.Hide(true);
   // Schedule a resurrection
   %this.Schedule($RespawnDelay, "Hide", false);
   %this.Schedule($RespawnDelay + 10, "StartFade", 3000, 0, false);
}
function Item::SchedulePop(%this)
{
   %this.Schedule($LoiterDelay - 1000, "StartFade", 3000, 0, true);
   %this.Schedule($LoiterDelay, "Delete");
}
function ItemData::OnThrow(%this,%user,%amount)
{
   // Remove the object from the inventory
   if (%amount $= "")
      %amount = 1;
   if (%this.maxInventory !$= "")
      if (%amount > %this.maxInventory)
         %amount = %this.maxInventory;
   if (!%amount)
      return 0;
   %user.DecInventory(%this,%amount);
   %obj = new Item() {
      datablock = %this;
      rotation = "0 0 1 " @ (GetRandom() * 360);
      count = %amount;
   };
   MissionGroup.Add(%obj);
   %obj.SchedulePop();
   return %obj;
}
function ItemData::OnPickup(%this,%obj,%user,%amount)
{
   %count = %obj.count;
```

```
   if (%count $= "")
      if (%this.maxInventory !$= "") {
         if (!(%count = %this.maxInventory))
            return;
      }
      else
         %count = 1;
   %user.IncInventory(%this,%count);
   if (%user.client)
      MessageClient(%user.client, 'MsgItemPickup', '\c0You picked up %1', %this.pickup-
Name);
   if (%obj.IsStatic())
      %obj.Respawn();
   else
      %obj.Delete();
   return true;
}
function ItemData::Create(%data)
{
   %obj = new Item() {
      dataBlock = %data;
      static = true;
      rotate = true;
   };
   return %obj;
}
datablock ItemData(Copper)
{
   category = "Coins";
   // Basic Item properties
   shapeFile = "~/data/models/items/kash1.dts";
   mass = 0.7;
   friction = 0.8;
   elasticity = 0.3;
   respawnTime = 30 * 60000;
   salvageTime = 15 * 60000;
   // Dynamic properties defined by the scripts
   pickupName = "a copper coin";
   value = 1;
};
datablock ItemData(Silver)
{
```

```
   category = "Coins";
   // Basic Item properties
   shapeFile = "~/data/models/items/kash100.dts";
   mass = 0.7;
   friction = 0.8;
   elasticity = 0.3;
   respawnTime = 30 * 60000;
   salvageTime = 15 * 60000;
   // Dynamic properties defined by the scripts
   pickupName = "a silver coin";
   value = 100;
};
datablock ItemData(Gold)
{
   category = "Coins";

   // Basic Item properties
   shapeFile = "~/data/models/items/kash1000.dts";
   mass = 0.7;
   friction = 0.8;
   elasticity = 0.3;
   respawnTime = 30 * 60000;
   salvageTime = 15 * 60000;
   // Dynamic properties defined by the scripts
   pickupName = "a gold coin";
   value = 1000;
};
datablock ItemData(FirstAidKit)
{
   category = "Health";
   // Basic Item properties
   shapeFile = "~/data/models/items/healthPatch.dts";
   mass = 1;
   friction = 1;
   elasticity = 0.3;
   respawnTime = 600000;
   // Dynamic properties defined by the scripts
   repairAmount = 200;
   maxInventory = 0; // No pickup or throw
};
function FirstAidKit::onCollision(%this,%obj,%col)
{
```

```
   if (%col.getDamageLevel() != 0 && %col.getState() !$= "Dead" )
   {
      %col.applyRepair(%this.repairAmount);
      %obj.respawn();
      if (%col.client)
      {
         messageClient
               (%col.client,'MSG_Treatment','\c2Medical treatment applied');
      }
   }
}
```

`$RespawnDelay` and `$LoiterDelay` are variables used to manage how long it takes to regenerate static items or how long they take to disappear when dropped.

After an item has been picked, if it is a static item, a new copy of that item will eventually be added to the game world using the `Item::respawn` method. The first statement in this method fades the object away, smoothly and quickly. Then the object is hidden, just to be sure. Finally, we schedule a time in the future to bring the object back into existence—the first event removes the object from hiding, and the second event fades the object in smoothly and slowly over a period of three seconds.

If we drop an item, we may want to have it removed from the game world to avoid object clutter (and concomitant bandwidth loss). We can use the `Item::schedulePop` method to make the dropped object remove itself from the world after a brief period of loitering. The first event scheduled is the start of a fade-out action, and after one second the object is deleted.

We can get rid of held items by throwing them using the `ItemData::onThrow` method. It removes the object from inventory, decrements the inventory count, creates a new instance of the object for inclusion in the game world, and adds it. It then calls the `SchedulePop` method just described to look after removing the object from the game world.

The `ItemData::onPickup` method is the one used by all items—it adds the item to the inventory and then sends a message to the client to indicate that the object has been picked up. If the object picked was a static one, it then schedules an event to add a replacement item into the world. If not, then the instance picked is deleted, and we see it no more.

The `ItemData::Create` method is the catchall object creation method for items. It creates a new data block based upon the passed parameter and sets the `static` and `rotate` properties to `true` before returning.

Next comes a collection of data blocks defining our coin and first-aid items. We will cover first-aid items more in Chapter 16.

The last method of interest is `FirstAidKit::onCollision`. This method will restore some health, by applying a repair value, to colliding objects if it needs it. Once the treatment has been applied, a message is sent to the client for display.

Running Emaga5

Once you've typed in all of the modules, you should be in a good position to test Emaga5. Table 5.3 shows the game key bindings that apply to in-game navigation.

Table 5.3 Emaga5 Game Key Bindings

Key	Description
up arrow	run forward
down arrow	run backward
left arrow	run (strafe) left
right arrow	run (strafe) right
numpad 0	jump and respawn
z	free look (hold key and move mouse)
tab	toggle player point of view
escape	quit game
tilde	open console
left mouse button	fire weapon

Figure 5.4 shows your player-avatar shortly after spawning in Emaga5.

Figure 5.4 The Avatar in Emaga5.

To test the game, travel around the world collecting gold, silver, and copper coins, and watch the total increase. You will have to watch out, though. The AI beasts will track you and shoot you if they spot you. Like the saying goes, you can run, but you'll only die tired! You *can* grab a crossbow and shoot back. In some of the huts you will find first-aid kits that will patch you up. One more thing—don't fall off cliffs. Not healthy.

As an exercise, investigate how you would enable a game timer to limit how much time you have to gather up the coins. Also, display a message if your score exceeds a certain value.

Have fun!

Moving Right Along

So, in this chapter you were exposed to more game structuring shenanigans—though nothing too serious. It's always a good idea to keep your software organized in ways that make sense according to the current state of the project. It just makes it that much easier to keep track of what goes where, and why.

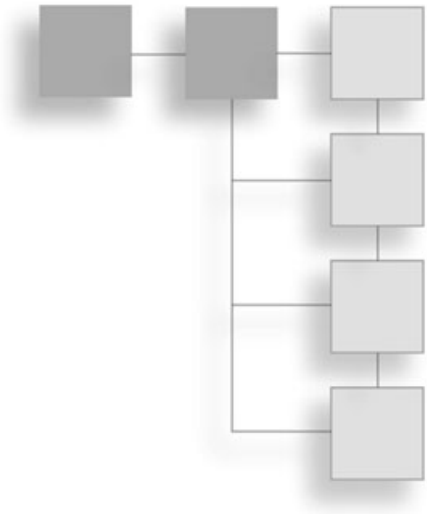
Then we looked at how we can add more features: splash screens, interfaces, and so on. You should be able to extrapolate from the small amount of game play stuff we added, like crossbows and pickable items, that the world really can be your oyster. What your game will do is limited only by your imagination.

In the next chapter, we'll poke a little deeper under the hood at one of the more hidden, yet very powerful capabilities that any decent game will need—messaging.

This page intentionally left blank

Team LRN

CHAPTER 6



NETWORK

Although little emphasis was given to the subject in recent chapters, a key feature of working with Torque is the fact that it was built around a client/server networking architecture.

Torque creates a `GameConnection` object, which is the primary mechanism that links the client (and the player) to the server. The `GameConnection` object is built from a `NetworkConnection` object. When the server needs to update clients, or when it receives updates from clients, the work is done through the good auspices of the `NetworkConnection`, and it is normally quite transparent at the game level.

What this means in practical terms is that the engine automatically handles things like movement and state changes or property changes of objects that populate a game world. Game programmers (like you and me) can then poke their grubby little fingers into this system to make it do their bidding without needing to worry about all the rest of the stuff, which Torque will manage—unless we decide to mess around with that too!

I know this seems a bit vague, so in this chapter we will attack the nitty-gritty so that you can really see how to use Torque's built-in networking to the best advantage.

First we will discuss the features, and look at examples of how they can be implemented, and then later in the chapter, after you update your Emaga sample program, you can try them out.

Direct Messaging

The quickest way to get down and dirty with the client/server networking in Torque is to use the `CommandToServer` and `CommandToClient` direct messaging functions. These extremely

Team LRN

useful "ad hoc" messaging functions are used for a wide variety of purposes in a Torque game, like in-game chat, system messages, and client/server synchronization.

CommandToServer

The `CommandToServer` function is used to send a message from a client to a server. Of course, the server needs to know that the message is coming and how to parse it to extract the data. The syntax is as follows:

CommandToServer(function [,arg1,...argn])

Parameters:	*function*	Message handler function on the server to be executed.
	arg1,...argn	Arguments for the function.
Return:	*nothing*	

An example of how to use the function would be a simple global chat macro capability where a player would press a key, and then a specific message would be broadcast to all other players. Here is how that would work:

First, we would bind a key combination to a specific function, say Ctrl+H bound to the function we'll call `SendMacro()`. In the key binding statement, we'll make sure to pass the value 1 as a parameter to `SendMacro()`.

`SendMacro()` could be defined on the client as this:

```
function SendMacro(%value)
{
  switch$ (%value)
  {
    case 1:
        %msg = "Hello World!";
    case 2:
        %msg = "Hello? Is this thing on?";
    default:
        %msg = "Nevermind!";
  }
  CommandToServer('TellEveryone', %msg);
}
```

So now, when the player presses Ctrl+H, the `SendMacro()` function is called, with its `%value` parameter set to 1. In `SendMacro()`, the `%value` parameter is examined by the `switch$` statement and sent to `case 1:`, where the variable `%msg` is stuffed with the string `"Hello World!"`. Then `CommandToServer` is called with the first parameter set to the tagged string `"TellEveryone"` and the second parameter set to our message.

Now here is where some of the Torque client/server magic elbows its way onto the stage. The client will already have a `GameConnection` to the server and so will already know where to send the message. In order to act on our message, the server side needs us to define the `TellEveryone` message handler, which is really just a special purpose function, something like this:

```
function ServerCmdTellEveryone(%client,%msg)
{
    TellAll(%client,%msg);
}
```

Notice the prefix `ServerCmd`. When the server receives a message from the client via the `CommandToServer()` function, it will look in its message handle list, which is a list of functions that have the `ServerCmd` prefix, and find the one that matches `ServerCmdTellEveryone`. It then calls that function, setting the first parameter to the `GameConnection` handle of the client that sent the message. It then sets the rest of the parameters to be the parameters passed in the message from the client, which in this case is `%msg` stuffed with the string `"Hello World!"`.

Then we can do what we want with the incoming message. In this case we want to send the message to all of the other clients that are connected to the server, and we'll do that by calling the `TellAll()` function. Now we *could* put the code right here in our `ServerCmdTellEveryone` message handler, but it is a better design approach to break the code out into its own independent function. We'll cover how to do this in the next section.

CommandToClient

Okay, here we are—we're the server, and we've received a message from a client. We've figured out that the message is the `TellEveryone` message, we know which client sent it, and we have a string that came along with the message. What we need to do now is define the `TellAll()` function, so here is what it could look like:

```
function TellAll( %sender, %msg)
{
   %count = ClientGroup.getCount();
   for ( %i = 0; %i < %count; %i++ )
   {
     %client = ClientGroup.getObject(%i);
      commandToClient(%client,'TellMessage', %sender, %msg);
    }
}
```

Our intention here is to forward the message to all of the clients. Whenever a client connects to the server, its `GameConnection` handle is added to the `ClientGroup`'s internal list. We

can use the ClientGroup's method getCount to tell us how many clients are connected. ClientGroup also has other useful methods, and one of them—the getObject method—will give us the GameConnection handle of a client, if we tell it the index number we are interested in.

If you want to test these example functions, I'll show you how to do that toward the end of the chapter. If you feel like giving it a go by yourself, I'll give you a small hint: The CommandToClient function is called from the *server* side, and the CommandToServer functions belong on the *client* side.

As you can see, CommandToClient is basically the server-side analogue to CommandToServer. The syntax is as follows:

CommandToClient(client, function [,arg1,...argn])

Parameters:	*client*	Handle of target client.
	function	Message handler function on the server to be executed.
	arg1,...argn	Arguments for the function.
Return:	*nothing*	

The primary difference is that although the client already knew how to contact the server when using CommandToServer, the same is not true for the server when using CommandToClient. It needs to know *which* client to send the message to each time it sends the message. So the simple approach is to iterate through the ClientGroup using the for loop, getting the handle for each client, and then sending each client a message using the CommandToClient() function, by specifying the client handle as the first parameter. The second parameter is the name of the message handler on the *client* side this time. Yup—works the same going that way as it did coming this way! Of course, the third parameter is the actual message to be passed.

So we need that message handler to be defined back over on the client. You can do it like this:

```
function clientCmdTellMessage(%sender, %msgString)
{
 // blah blah blah
}
```

Notice that when we called this function there were four parameters, but our definition only has two in the parameter list. Well, the first parameter was the client handle, and because we are on the client, Torque strips that out for us. The second parameter was the message handler identifier, which was stripped out after Torque located the handler function and sent the program execution here. So the next parameter is the sender, which is the client that started this whole snowball rolling, way back when. The last parameter is, finally, the actual message.

Team LRN

I'll leave it up to you to decide what to do with the message. The point here was to show this powerful messaging system in operation. You can use it for almost anything you want.

Direct Messaging Wrap-up

`CommandToServer` and `CommandToClient` are two sides of the same direct messaging coin and give us, as game programmers, a tremendous ability to send messages back and forth between the game client and the game server.

Direct messaging can also be an important tool in the fight against online cheating in your game. You can, in theory and in practice, require all user inputs to go to the server for approval before executing any code on the client. Even things like changing setup options on the client—which are not normally the sort of thing that servers would control—can be easily programmed to require server control using the technique we just looked at.

The actual amount of server-side control you employ will be dictated by both available bandwidth and server-side processing power. There is a lot that can be done, but it is a never-ending series of tradeoffs to find the right balance.

Triggers

Right off the bat, there is potential for confusion when discussing the term trigger in Torque, so let's get that out of the way. There are four kinds of triggers that people talk about when programming with Torque:

- area triggers
- animation triggers
- weapon state triggers
- player event control triggers

I'll introduce you to all four here but we'll talk about three of them—area triggers, animation triggers, and weapon state triggers—in more detail in future chapters.

Area Triggers

Area triggers are a special in-game construct. An area in the 3D world of a game is defined as a *trigger object*. When a player's avatar enters the bounds of the trigger area, an event message is posted on the server. We can write handlers to be activated by these messages. We will be covering area triggers in more depth in Chapter 22.

Animation Triggers

Animation triggers are used to synchronize footstep sounds with walking animation in player models. Modeling tools that support animation triggers have ways of tagging frames

of animation sequences. The tags tell the game engine that certain things should happen when this frame of an animation is being displayed. We'll discuss these later in Chapter 20.

Weapon State Triggers

Torque uses weapon state triggers for managing and manipulating weapon states. These triggers dictate what to do when a weapon is firing, reloading, recoiling, and so on. We'll look at this in more detail later in Chapter 20 in the section "Weapon Sounds".

Player Event Control Triggers

Finally, there are *player event control triggers*, which are a form of indirect messaging of interest to us in this chapter. These mechanisms are used to process certain player inputs on the client in real time. You can have up to six of these triggers, each held by a variable with the prefix `$mvTriggerCountn` (where *n* is an index number from 0 to 5).

When we use a trigger move event, we increment the appropriate `$mvTriggerCountn` variable on the client side. This change in value causes an update message back to the server. The server will process these changes in the context of our control object, which is usually our player's avatar. After the server acts on the trigger, it decrements its count. If the count is nonzero, it acts again when it gets the next change in its internal scheduling algorithm. In this way we can initiate these trigger events by incrementing the variable as much as we want (up to a maximum of 255 times), without having to wait and see if the server has acted on the events. They are just automatically queued up for us via the `$mvTriggerCountn` variable mechanism.

Torque has default support for the first four control triggers built into its player and vehicle classes (see Table 6.1).

Table 6.1 Default Player Event Control Triggers

Trigger	Default Action
`$mvTriggerCount0`	Shoots or activates the mounted weapon in image slot 0 of the player's avatar. (The "fire" button, so to speak.)
`$mvTriggerCount1`	Shoots or activates the mounted weapon in image slot 1 of the player's avatar. (The "alt fire".)
`$mvTriggerCount2`	Initiates the "jump" action and animation for the player's avatar.
`$mvTriggerCount3`	Initiates the "jetting" (extra boost) action and animation for the vehicle on which a player's avatar is mounted.
`$mvTriggerCount4`	Unassigned.
`$mvTriggerCount5`	Unassigned.

Team LRN

In the server control code, we can put a trigger handler in our player's avatar for any of these triggers that override the default action. We define a trigger handler like this:

```
function MyAvatarClass::onTrigger(%this, %obj, %triggerNum, %val)
{
   // trigger activity here
  $switch(%triggerNum)
  {
    case 0:
      //replacement for the "fire" action.
    case 1:
      //replacement for the "alt fire" action.
    case 2:
      //replacement for the "jump" action.
    case 3:
      //replacement for the "jetting" action.
    case 4:
      //whatever you like
    case 5:
      //whatever you like
  }
}
```

The MyAvatarClass class is whatever you have defined in your player avatar's datablock using the following statement:

```
   className = MyAvatarClass;
```

To use these handlers, you merely have to increment one of the player event control triggers on the client, something like this:

```
function mouseFire(%val)
{
   $mvTriggerCount0++;
}
```

GameConnection Messages

Most of the other kinds of messaging used when making a game with Torque are handled automatically. However, in addition to the direct messaging techniques we just looked at, there are other more indirect messaging capabilities available to the Torque game developer. These are messages related to the `GameConnection` object.

I call these methods *indirect* because we, as programmers, don't get to use them in any old way of our choosing. But we *can*, nonetheless, use these methods, in the form of message handlers, when the Torque Engine decides it needs to send the messages.

What GameConnection Messages Do

`GameConnection` messages are of great importance to us during the negotiation process that takes place between the client and server when a client joins a game. They are network messages with game-specific uses, as opposed to being potentially more general-purpose network messages.

Torque calls a number of `GameConnection` message handlers at different times during the process of establishing, maintaining, and dropping game-related connections. In the Torque demo software, many of these handlers are defined in the common code base, whereas others aren't used at all. You are encouraged to override the common code message handlers with your own `GameConnection` message handlers or use the unused handlers, if you need to.

Specifics

During program execution, the client will at some point try to connect to the server using a set of function calls like this:

```
%conn = new GameConnection(ServerConnection);
%conn.SetConnectArgs(%username);
%conn.Connect();
```

In this example the `%conn` variable holds the handle to the `GameConnection`. The `Connect()` function call initiates a series of network transactions that culminate at the server with a call to the `GameConnection::OnConnect` handler.

The following descriptions are listed roughly in the order that they are used in the program.

onConnectionRequest()

Parameters:	*none*	
Return:	*"" (null string)*	Indicates that connection is accepted.
	None	Indicates rejection for some reason.
Description:	Called when a client attempts a connection, before the connection is accepted.	
Usage:	Common—Server	

This handler is used to check if the server-player capacity has been exceeded. If not exceeded, then "" is returned, which allows the connection process to continue. If the server is full, then `CR_SERVERFULL` is returned. Returning any value other than "" will cause an error condition to be propagated back through the engine and sent to the client as a call

to the handler `GameConnection:: onConnectRequestRejected`. Any arguments that were passed to `GameConnection::::Connect` are also passed to this handler by the engine.

onConnectionAccepted(handle)

Parameters:	*handle*	`GameConnection` handle.
Return:	*nothing*	
Description:	Called when a `Connect` call succeeds.	
Usage:	Client	

This handler is a good place to make last-minute preparations for a connected session.

onConnectRequestRejected(handle, reason)

Parameters:	*handle*	`GameConnection` handle.
	reason	Indicates why connection was rejected.
Return:	*nothing*	
Description:	Called when `Connect` call fails.	
Usage:	Client	

If you arrive in this handler you should display, or at least log, the fact that the connection was rejected and why.

onConnect(client, name)

Parameters:	*client*	Client's `GameConnection` handle.
	name	Name of client's account or username.
Return:	*nothing*	
Description:	Called when a client has successfully connected.	
Usage:	Server	

In this case the second parameter (`%name`) is the value the client has used, while establishing the connection, as the parameter to the `%(GameConnection).SetConnectArgs(%username)` call.

onConnectRequestTimedOut(handle)

Parameters:	*handle*	`GameConnection` handle.
Return:	*nothing*	
Description:	Called when establishing a connection takes too long.	
Usage:	Client	

When this gets called you probably want to display, or at least log, some message indicating that the connection has been lost because of a timeout.

Team LRN

onConnectionTimedOut(handle)

Parameters:	*handle*	`GameConnection` handle.
Return:	*nothing*	
Description:	Called when a connection ping (heartbeat) has not been received.	
Usage:	Server, Client	

When this gets called you probably want to display, or at least log, some message indicating that the connection has been lost because of a timeout.

onConnectionDropped(handle, reason)

Parameters:	*handle*	`GameConnection` handle.
	reason	String indicating why server dropped the connection.
Return:	*nothing*	
Description:	Called when the server initiates the disconnection of a client.	
Usage:	Client	

When this gets called you probably want to display, or at least log, some message indicating that the connection has been lost because of a timeout.

onConnectRequestRejected(handle, reason)

Parameters:	*handle*	`GameConnection` handle.
	reason	See Table 6.2 for a list of conventional reason codes defined by GarageGames in script.
Return:	*nothing*	
Description:	Called when a client's connection request has been turned down by the server.	
Usage:	Client	

When this gets called you probably want to display, or at least log, some message indicating that the connection has been lost because of a timeout.

onConnectionError(handle, errorString)

Parameters:	*handle*	`GameConnection` handle.
	errorString	String indicating the error encountered.
Return:	*nothing*	
Description:	General connection error, usually raised by ghosted objects' initialization problems, such as missing files. The errorString is the server's connection error message.	
Usage:	Client	

Team LRN

Table 6.2 Connection Request Rejection Codes

Reason Code	Meaning
CR_INVALID_PROTOCOL_VERSION	The wrong version of client was detected.
CR_INVALID_CONNECT_PACKET	There is something wrong with the connection packet.
CR_YOUAREBANNED	Your game username has been banned.
CR_SERVERFULL	The server has reached the maximum number of players.
CHR_PASSWORD	The password is incorrect.
CHR_PROTOCOL	The game protocol version is not compatible.
CHR_CLASSCRC	The game class version is not compatible.
CHR_INVALID_CHALLENGE_PACKET	The client detected an invalid server response packet.

onDrop(handle, reason)

Parameters:	*handle*	`GameConnection` handle.
	reason	Reason for connection being dropped, passed from server.
Return:	*nothing*	
Description:	Called when a connection to a server is arbitrarily dropped.	
Usage:	Client	

initialControlSet (handle)

Parameters:	*handle*	`GameConnection` handle.
Return:	*nothing*	
Description:	Called when the server has set up a control object for the `GameConnection`. For example, this could be an avatar model or a camera.	
Usage:	Client	

setLagIcon(handle, state)

Parameters:	*handle*	`GameConnection` handle.
	state	Boolean that indicates whether to display or hide the icon.
Return:	*nothing*	
Description:	Called when the connection state has changed, based upon the lag setting. *state* is set to `true` when the connection is considered temporarily broken or set to `false` when there is no loss of connection.	
Usage:	Client	

onDataBlocksDone(handle, sequence)

Parameters:	*handle*	`GameConnection` handle.
	sequence	Value that indicates which set of data blocks has been transmitted.
Return:	*nothing*	
Description:	Called when the server has received confirmation that all data blocks have been received.	
Usage:	Server	

Use this handler to manage the mission loading process and any other activity that transfers datablocks.

onDataBlockObjectReceived(index, total)

Parameters:	*index*	Index number of data block objects.
	total	How many sent so far.
Return:	*nothing*	
Description:	Called when the server is ready for data blocks to be sent.	
Usage:	Client	

onFileChunkReceived(file, ofs, size)

Parameters:	*file*	The name of the file being sent.
	ofs	Offset of data received.
	size	File size.
Return:	*nothing*	
Description:	Called when a chunk of file data from the server has arrived.	
Usage:	Client	

onGhostAlwaysObjectReceived()

Parameters:	*none*
Return:	*nothing*
Description:	Called when a ghosted object's data has been sent across from the server to the client.
Usage:	Client

Team LRN

onGhostAlwaysStarted(count)

Parameters:	*count*	The number of ghosted objects dealt with so far.
Return:	*nothing*	
Description:	Called when a ghosted object has been sent to the client.	
Usage:	Client	

Finding Servers

When you offer a game with networked client/server capabilities, there needs to be some means for players to find servers to which to connect. On the Internet, a fairly widely implemented technique is to employ a *master server*. The master server's job is fairly straightforward and simple. It keeps a list of active game servers and provides a client with the necessary information to connect to any one of the servers if desired.

To see the utility of such a simple system, just take a look at NovaLogic, makers of the successful *Delta Force* series of first-person shooters. NovaLogic still hosts master servers for customers who bought the original *Delta Force* games from the late 1990s! The overhead of such a simple system is minimal, and the benefit in customer good will is tremendous.

The *Tribes* series of games, upon which Torque is based, also offers such master servers, as do many other games out there.

On a small- to medium-sized local area network, this is not too onerous a task—an extremely simple method is to have the client merely examine a specified port on all visible nodes to see if a server is present, and that's what we're going to be doing in this chapter.

Code Changes

We are going to implement "find a server" support in our version of Emaga for this chapter. We will create Emaga6 by modifying Emaga5, the game from the last chapter.

First, copy your entire C:\Emaga5 folder to a new folder, called C:\Emaga6. Then, for the sake of clarity, rename the UltraEdit project file to chapter6.prj. Now open your new Chapter 6 UltraEdit project. All changes will be made in the control code. In addition to changes to the actual program code, you might want to also change any Chapter 5 comment references so they refer to Chapter 6—it's your call.

Client—Initialize Module

We'll make our first change in control/client/initialize.cs. Open that module and locate the function `InitializeClient`. Add the following statements to the very beginning of the function:

```
$Client::GameTypeQuery = "3DGPAI1";
$Client::MissionTypeQuery = "Any";
```

When one of our servers contacts the master server, it uses the variable `$Client::GameTypeQuery` to filter out game types that we aren't interested in. For your game, you can set any game type you like. Here we are going to go with 3DGPAI1 because there will be at least one 3DGPAI1 server listed on the master server, and for the purpose of illustration it is better to see one or two 3DGPAI1servers listed than nothing at all. You can change this later at your leisure.

The variable `$Client::MissionTypeQuery` is used to filter whatever specific game play styles are available. By specifying `Any`, we will see any types that are available. This is also something we can define in whatever way we want for our game.

Farther down will be a call to `InitCanvas`. Although it is not really important to make the master server stuff work, change that statement to this:

```
InitCanvas("emaga6 - 3DGPAi1 Sample Game");
```

Doing so reflects the fact that we are now in Chapter 6 and not in Chapter 5 anymore.

Next, there are a series of calls to `Exec`. Find the one that loads playerinterface.gui, and put the following line after that one:

```
Exec("./interfaces/serverscreen.gui");
```

Then find the call to `Exec` that loads screens.cs, and add the following statement after it:

```
Exec("./misc/serverscreen.cs");
```

Finally, toward the end of the function, find the `Exec` call that loads connections.cs. After that statement, and before the call to `Canvas.SetContent`, add the following statement:

```
SetNetPort(0);
```

This statement is critical. Although we will never use port 0, it is necessary to make this call to ensure that the TCP/IP code in Torque works correctly. Later on in other modules the appropriate port will be set, depending on what we are doing.

New Modules

More typing! But not as much as in previous chapters, so don't fret. We have to add a new interface module and a module to contain the code that manages its behavior.

Client—ServerScreen Interface Module

Now we have to add the `ServerScreen` *interface* module. This module defines buttons, text labels, and a scroll control that will appear on the screen; we can use it to query the

master server and view the results. Type in the following code and save it as control/client/interfaces/serverscreen.gui.

```
//==============================================================================
// control/client/interfaces/serverscreen.gui
//
//  Server query interface module for 3DGPAI1 emaga6 sample game
//
//  Copyright (c) 2003 by Kenneth C.  Finney.
//==============================================================================
new GuiChunkedBitmapCtrl(ServerScreen) {
   profile = "GuiContentProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   bitmap = "./emaga_background";
   useVariable = "0";
   tile = "0";

   new GuiControl() {
      profile = "GuiWindowProfile";
      horizSizing = "center";
      vertSizing = "center";
      position = "100 100";
      extent = "600 300";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";

      new GuiTextCtrl() {
         profile = "GuiTextProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "15 40";
         extent = "30 20";
         minExtent = "8 8";
         visible = "1";
         helpTag = "0";
         text = "Pass";
```

```
      maxLength = "255";
   };
   new GuiButtonCtrl(JoinServer) {
      profile = "GuiButtonProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "318 272";
      extent = "127 23";
      minExtent = "8 8";
      visible = "1";
      command = "Canvas.getContent().Join();";
      helpTag = "0";
      text = "Connect";
         active = "0";
   };
   new GuiScrollCtrl() {
      profile = "GuiScrollProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "10 75";
      extent = "437 186";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      willFirstRespond = "1";
      hScrollBar = "dynamic";
      vScrollBar = "alwaysOn";
      constantThumbHeight = "0";
      defaultLineHeight = "15";
      childMargin = "0 0";

      new GuiTextListCtrl(ServerList) {
         profile = "GuiTextArrayProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "0 0";
         extent = "419 8";
         minExtent = "8 8";
         visible = "1";
         helpTag = "0";
         enumerate = "0";
         resizeCell = "1";
```

```
      columns = "0 40 195 260 325 385";
      fitParentWidth = "1";
      clipColumnText = "0";
         noDuplicates = "false";
   };
};
new GuiTextEditCtrl() {
   profile = "GuiTextEditProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "98 15";
   extent = "134 16";
   minExtent = "8 8";
   visible = "1";
   variable = "Pref::Player::Name";
   helpTag = "0";
   maxLength = "255";
   historySize = "0";
   password = "0";
   tabComplete = "0";
};
new GuiTextCtrl() {
   profile = "GuiTextProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "12 11";
   extent = "79 20";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
   text = "Player Name:";
   maxLength = "255";
};
new GuiTextCtrl() {
   profile = "GuiTextProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "269 42";
   extent = "44 20";
   minExtent = "8 8";
   visible = "1";
   helpTag = "0";
```

```
      text = "Players";
      maxLength = "255";
   };
   new GuiTextCtrl() {
      profile = "GuiTextProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "335 42";
      extent = "44 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "Version";
      maxLength = "255";
   };
   new GuiTextCtrl() {
      profile = "GuiTextProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "412 42";
      extent = "35 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "Game";
      maxLength = "255";
   };
   new GuiTextCtrl() {
      profile = "GuiTextProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "212 42";
      extent = "26 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "Ping";
      maxLength = "255";
   };
   new GuiTextCtrl() {
      profile = "GuiTextProfile";
      horizSizing = "right";
```

```
      vertSizing = "bottom";
      position = "72 42";
      extent = "74 20";
      minExtent = "8 8";
      visible = "1";
      helpTag = "0";
      text = "Server";
      maxLength = "255";
   };
   new GuiButtonCtrl() {
      profile = "GuiButtonProfile";
      horizSizing = "right";
      vertSizing = "top";
      position = "10 272";
      extent = "127 23";
      minExtent = "8 8";
      visible = "1";
      command = "Canvas.getContent().Close();";
      helpTag = "0";
      text = "Close";
   };
   new GuiControl(QueryStatus) {
      profile = "GuiWindowProfile";
      horizSizing = "center";
      vertSizing = "center";
      position = "72 129";
      extent = "310 50";
      minExtent = "8 8";
      visible = "0";
      helpTag = "0";

      new GuiButtonCtrl(CancelQuery) {
         profile = "GuiButtonProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "9 15";
         extent = "64 20";
         minExtent = "8 8";
         visible = "1";
         command = "Canvas.getContent().Cancel();";
         helpTag = "0";
         text = "Cancel Query";
```

```
         };
         new GuiProgressCtrl(StatusBar) {
            profile = "GuiProgressProfile";
            horizSizing = "right";
            vertSizing = "bottom";
            position = "84 15";
            extent = "207 20";
            minExtent = "8 8";
            visible = "1";
            helpTag = "0";
         };
         new GuiTextCtrl(StatusText) {
            profile = "GuiProgressTextProfile";
            horizSizing = "right";
            vertSizing = "bottom";
            position = "85 14";
            extent = "205 20";
            minExtent = "8 8";
            visible = "1";
            helpTag = "0";
            maxLength = "255";
         };
      };
   };
};
```

The first half of the module is an interface definition, defining a number of buttons, text labels, and a scroll control that will appear on the screen. Most of the properties and control types have been covered in previous chapters; however, some of them are of particular note here.

The first item of interest is the `GuiScrollCtrl`. This control provides a scrollable vertical list of records; in this case it will be a list of servers that satisfy the filters used in subsequent `Query` calls that we will look at a bit later.

Some of the `GuiScrollCtrl` properties of interest are explained in Table 6.3.

The next significant control to examine is the `GuiTextEditCtrl`. It has an interesting property, shown by this statement:

```
variable = "Pref::Player::Name";
```

What this does is display the contents of the variable `Pref::Player::Name` in the control's content. If we change that content by placing our edit cursor in the control's field while it

Table 6.3 Selected GuiScrollCtrl Properties

Property	Description
`willFirstRespond`	If set to `true` or 1, indicates that this control will respond to user inputs first, before passing them on to other controls.
`hScrollBar`	Indicates how to decide whether to display the horizontal scroll bar. The choices are: `alwaysOn`: The scroll bar is always visible. `alwaysOff`: The scroll bar is never visible. `dynamic`: The scroll bar is not visible until the number of records in the list exceeds the number of lines available to display them. If this happens the scroll bar is turned on and made visible.
`vScrollBar`	The same as `hScrollBar` but applies to the vertical scroll bar.
`constantThumbHeight`	Indicates whether the *thumb*, the small rectangular widget in the scroll bar that moves as you scroll, will have a size that is proportional to the number of entries in the list (the longer the list, the smaller the thumb) or will have a constant size. Setting this property to 1 ensures a constant size; 0 ensures proportional sizing.

is being displayed and typing in new text, then the contents of the variable `Pref::Player::Name` are also changed.

Also in this `GuiTextEditCtrl` control is the following statement:

```
historySize = "0";
```

This control has the ability to store a history of previous values that were held in the control's edit box. We can scroll through the list's previous values by pressing the Up Arrow and Down Arrow keys. This property sets the maximum number of values that can be saved in the control's history. A setting of 0 means that no history will be saved.

Now go take a look at the control of type `GuiControl` with the name `QueryStatus`. This is the definition of a subscreen that will display the progress of the query. It contains other controls that we've seen before, but I just want you to note how they are nested within this control, which is nested within the larger `ServerScreen`.

Client—ServerScreen Code Module

Next, we will add the `ServerScreen` *code* module. This module defines how the `ServerScreen` interface module will behave. Type in the following code and save it as control/client/misc/serverscreen.cs.

```
//==============================================================================
// control/client/misc/serverscreen.cs
//
```

```
// Server query code module for 3DGPAI1 emaga6 sample game
//
// Copyright (c) 2003 by Kenneth C.  Finney.
//=============================================================================
function ServerScreen::onWake()
{
   JoinServer.SetActive(ServerList.rowCount() > 0);
   ServerScreen.queryLan();
}


function ServerScreen::QueryLan(%this)
{
   QueryLANServers(
      28000,      // lanPort for local queries
      0,          // Query flags
      $Client::GameTypeQuery,       // gameTypes
      $Client::MissionTypeQuery,    // missionType
      0,          // minPlayers
      100,        // maxPlayers
      0,          // maxBots
      2,          // regionMask
      0,          // maxPing
      100,        // minCPU
      0           // filterFlags
      );
}

function ServerScreen::Cancel(%this)
{
   CancelServerQuery();
}

function ServerScreen::Close(%this)
{
   CancelServerQuery();
   Canvas.SetContent(MenuScreen);
}

function MasterScreen::Update(%this)
{
   QueryStatus.SetVisible(false);
```

```
   ServerList.Clear();
   %sc = GetServerCount();
   for (%i = 0; %i < %sc; %i++)
   {
      SetServerInfo(%i);
      ServerList.AddRow(%i,
         ($ServerInfo::Password? "Yes": "No") TAB
         $ServerInfo::Name TAB
         $ServerInfo::Ping TAB
         $ServerInfo::PlayerCount @ "/" @ $ServerInfo::MaxPlayers TAB
         $ServerInfo::Version TAB
         $ServerInfo::GameType TAB
         %i);
   }
   ServerList.Sort(0);
   ServerList.SetSelectedRow(0);
   ServerList.ScrollVisible(0);
   JoinServer.SetActive(ServerList.RowCount() > 0);
}

function ServerScreen::Join(%this)
{
   CancelServerQuery();
   %id = ServerList.GetSelectedId();
   %index = GetField(ServerList.GetRowTextById(%id),6);
   if (SetServerInfo(%index)) {
      %conn = new GameConnection(ServerConnection);
      %conn.SetConnectArgs($pref::Player::Name);
      %conn.SetJoinPassword($Client::Password);
      %conn.Connect($ServerInfo::Address);
   }
}

function onServerQueryStatus(%status, %msg, %value)
{
   if (!QueryStatus.IsVisible())
      QueryStatus.SetVisible(true);

   switch$ (%status) {
      case "start":

      case "ping":
```

```
            StatusText.SetText("Ping Servers");
            StatusBar.SetValue(%value);

         case "query":

         case "done":
            QueryStatus.SetVisible(false);
            Screen.Update();
      }
}
```

This module is where we've put the code that controls how the Master Server screen behaves.

The first function, `ServerScreen::onWake`, defines what to do when the screen is displayed. In this case we first set the Join button to be active if there are any servers in the server list at the moment we display the screen. Then, `MasterScreen::QueryLAN`, is called. It executes a call to `QueryLANServers`, which reaches out across the local area network and talks to each computer on port 28000 (you can use any available port). If it manages to contact a computer with a game server running on that port, it establishes contact with the game server, obtains some information from it, and adds that server to a list. There are quite a few parameters to the call to `QueryLANServers`. The following syntax definition shows them in more detail:

QueryLANServers (port, flags,gtype,mtype,minplayers,maxplayers,maxbots, region,ping,cpu,filters,buddycount, buddylist)

Parameters:	*port*	The TCP/IP port where game servers are expected to be found.
	flags	Query flags. Choices: 0×00 = online query 0×01 = offline query 0×02 = no string compression
	gtype	Game type string
	mtype	Mission type string
	minplayers	Minimum number of players for viable game
	maxplayers	Maximum allowable players
	maxbots	Maximum allowable connected AI bots
	region	Numeric discriminating mask
	ping	Maximum ping for connecting clients; 0 means no maximum
	mincpu	Minimum specified CPU capability

continued

	filterflags	Server filters. Choices: 0×00 = dedicated 0×01 = not password protected 0×02 = Linux 0×80 = current version
	buddycount	Number of buddy servers in buddy list
	buddylist	List of server names that are buddies to this server
Return:	*nothing*	

The response to the `QueryLANServers` function is accessible from the `ServerList` array.

The next function, `ServerScreen::Cancel`, is called when the Cancel button is pressed while the query is under way.

After that is the `ServerScreen::Close` function, which is called when the user presses the Close button. It cancels any pending query and then returns to the `MenuScreen`.

`ServerScreen::Update` is the function that inserts the obtained information in the `ServerList` after it is obtained from the master server. The information is found in the `$ServerInfo` array. To update the scrolling display, we find the number of servers that pass the filters on the master by calling `GetServerCount`. Then we iterate through our displayable list, extracting the fields from each `$ServerInfo` record. Take note of the call to `SetServerInfo`. Passing an index number to this function sets the `$ServerInfo` array to point to a specific record in the `MasterServerList`. Then we access the individual fields in the `$ServerInfo` array by referencing them with the colon operator: `$ServerInfo::Name` or `$ServerInfo::Name`, to demonstrate with two examples.

The next function, `ServerScreen::Join`, defines how we go about joining a server that has been selected from the list. First, we cancel any outstanding queries, get the handle of the server record that is highlighted in the interface, and then use that to obtain the index number of the server record. We use the `SetServerInfo` to set the `$ServerInfo` array to point to the right server record, and then we can access the values. After setting some network parameters, we finally use `$ServerInfo::Address` to make the network connection.

The last function in the module is the message handler callback that makes the whole shebang go: `onServerQueryStatus`. It gets called repeatedly as the server query process unfolds. We use the `%status` variable to determine what response we are receiving from the master server, and then we use either the `%msg` or `%value` variable, set by the master server to update various fields in the displayed server list. The start and query cases aren't needed in our example.

Dedicated Server

Sometimes we will want to host a game as a server without having to bother with a graphical user interface. One reason we might want to do this is because we want to run the server on a computer that doesn't have a 3D accelerated graphics adapter. Another reason is because we might want to test our client/server connectivity and master server query capabilities. This need arises because we can't run two instances of the Torque graphical client at the same time. However, if we have the ability to run as a dedicated server, we can run multiple dedicated servers, while running one instance of the graphical client, all on the same computer. And if we have set up the dedicated servers appropriately, other players out on the network can connect to our servers.

There are a few more modules you will have to change to implement the dedicated server capabilities.

Root Main Module

In this module we'll need to add some command line switches in case we want to use the command line interface of Windows, or we'll need to we decide to embed the switches in a Windows shortcut. Either of these methods is how we can tell the game to run the server in dedicated mode. In the module main.cs located in the *root game folder* (which is the folder where the tge.exe executable is located for your Chapter 6 version of Emaga), locate the `ParseArgs` function, and scroll down until you find the statement containing `$switch($currentarg)`. Type the following code in directly after the `$switch` statement:

```
case "-dedicated":
  $Server::Dedicated = true;
  EnableWinConsole(true);
  $argumentFlag[$i]++;

case "-map":
  $argumentFlag[$i]++;
  if ($nextArgExists)
  {
     $mapArgument = $nextArgument;
     $argumentFlag[$i+1]++;
     $i++;
  }
  else
     Error("Error: Missing argument. Usage: -mission <filename>");
```

Both of these switches are needed to run a dedicated server. The `-dedicated` switch puts us into the right mode, and then the `-map` switch tells us which mission map to load when the server first starts running.

The result of these changes is that we can now invoke the dedicated server mode by launching the game with the following syntax from the command line (don't try it yet): tge.exe -dedicated -map control/data/maps/book_ch6.mis.

The game will launch, and all you will see will be a console window. You will be able to type in console script statements, just as you can when you use the tilde ("~") key in the graphical client interface. However, don't try this just yet, because we still need to add the actual dedicated server code!

You can also create a shortcut to the tge.exe executable and modify the Target box in the shortcut properties to match the command line syntax above. Then you can launch the server merely by double-clicking on the shortcut icon.

Control—Main Module

Next, we have a quick modification to make to control/main.cs. In the `OnStart` function, locate the line that contains `InitializeClient`. Replace that one line with these four lines:

```
if ($Server::Dedicated)
   InitializeDedicatedServer();
else
   InitializeClient();
```

Now, when the program detects that the `-dedicated` switch was used, as described in the previous section, it will fire up in dedicated mode, not in client mode.

Control—Initialize Module

Okay, the meat of the dedicated server code is contained in this module. Open up the module control/server/initialize.cs and type in the following lines just before the `InitializeServer` function.

```
$pref::Master0 = "2:master.garagegames.com:28002";
$Pref::Server::ConnectionError = "You do not have the correct version of 3DGPAI1 client
or the related art needed to play on this server. This is the server for Chapter 6.
Please check that chapter for directions.";
$Pref::Server::FloodProtectionEnabled = 1;
$Pref::Server::Info = "3D Game Programming All-In-One by Kenneth C. Finney.";
$Pref::Server::MaxPlayers = 64;
$Pref::Server::Name = "3DGPAI1 Book - Chapter 6 Server";
$Pref::Server::Password = "";
$Pref::Server::Port = 28000;
$Pref::Server::RegionMask = 2;
$Pref::Server::TimeLimit = 20;
$Pref::Net::LagThreshold = "400";
```

```
$pref::Net::PacketRateToClient = "10";
$pref::Net::PacketRateToServer = "32";
$pref::Net::PacketSize = "200";
$pref::Net::Port = 28000;
```

You can change the string values to be anything you like as long as it suits your purposes. You should leave the `RegionMask` as is for now.

Next, locate the function `InitializeServer` again, and insert the following lines at the very beginning of the function:

```
$Server::GameType = "3DGPAI1";
$Server::MissionType = "Emaga6";
$Server::Status = "Unknown";
```

This value will be updated when the server makes contact with the master server.

Finally, you will need to add this entire function to the end of the module:

```
function InitializeDedicatedServer()
{
   EnableWinConsole(true);
   Echo("\n--------- Starting Dedicated Server ---------");

   $Server::Dedicated = true;

   if ($mapArgument !$= "") {
      CreateServer("MultiPlayer", $mapArgument);
   }
   else
      Echo("No map specified (use -map <filename>)");
}
```

This function enables the Windows console, sets the dedicated flag, and then calls `CreateServer` with the appropriate values. Now it may not do very much and therefore seem to be not too necessary, but the significance with the `InitializeDedicatedServer` function is in what it *doesn't* do compared with the `InitializeClient` function, which would have otherwise been called. So that's the reason why it exists.

Testing Emaga6

With all of the changes we've made here, we're going to want to see it run. It's really fairly easy. Open a command shell in Windows, and change to the folder where you've built the code for this chapter's program. Then run the dedicated server by typing in this command: tge.exe -dedicated -map control/data/maps/book_ch6.mis.

Team LRN

After it displays lots of start-up information, it will eventually settle down and tell you in the console window that it has successfully loaded a mission. When you see these things, your dedicated server is running fine.

tip

You may be wondering how to do this over the Internet. I've written a different version of this chapter that is available on the Internet as a supplement called "Internet Game Hosting". You can find it at http://cerdipity.no-ip.com/cerdipity and look for the 3DGPAI1 link on the left-hand side, near the top.

Next, double-click your tge.exe icon as you've done in the past to run the Emaga client. When the Menus screen appears, click the Connect To Server button. Look for the 3DGAPI1 server name (or whatever value you assigned to `$Pref::Server::Name` in the Control—Initialize module). Select that server entry, and then click Join. Watch the progress bars, and eventually you will find yourself deposited in the game. Send copies of this to your friends and get them to join in for some freewheeling havoc or reckless mayhem—whichever you prefer!

If you will recall, back at the beginning of the chapter, in the "Direct Messaging" section, we discussed the functions `CommandToClient` and `CommandToServer`. You might want to take this opportunity to test the code shown in that section. Put the `SendMacro` function in your C:\emagaCh6\control\client\misc\presetkeys.cs module, and then add the `ServerCmdTellEveryone` and `TellAll` functions to the end of your C:\emagaCh6\control\server\server.cs module. You can go ahead and test it now, if you like.

Moving Right Along

Now you have some understanding how to pass messages back and forth between the client and the server. Keep in mind when you contemplate these things that there can be many clients—hockey socks full of clients, even. There will probably only be one server but you are in no way restricted to only one server. It's all a matter of programming.

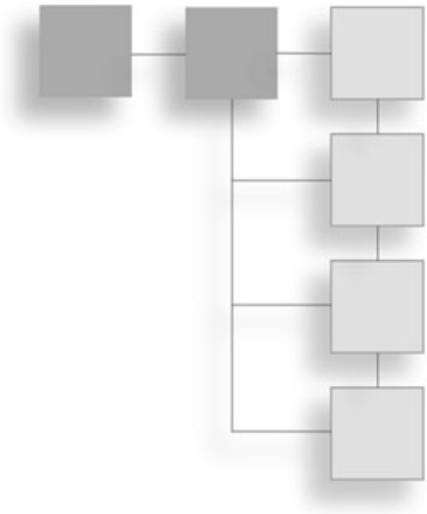
You've also seen how you can track specific clients on the server via their `GameConnections`. As long as you know the handle of the client, you can access any of that client's data.

In the next chapter, we'll poke our noses into the Common code that we have been shying away from. We want to do this so that we can get a better "big picture" understanding of how our game can operate.

This page intentionally left blank

Team LRN

CHAPTER 7



Common Scripts

For the last several chapters I have been keeping the contents of the common code folder tree out of the limelight. I hope you haven't started thinking that it is some deep dark keep-it-in-the-family-only secret, because it isn't. The reason for maintaining the obscurity is because we've been looking at the areas of scripting that you will most likely want to change to suit your game development needs, and that means stuff *not* in the common code.

Having said that, there may be areas in the common code that you will want to customize or adjust in one way or another. To that end, we are going to spend this chapter patrolling the common code to get the lay of the land.

You can gain access to this code for yourself in the common folder tree of any of the Emaga versions you installed in the previous chapters.

Game Initialization

As you may recall from earlier chapters, the common code base is treated as if it was just another add-on or Mod. It is implemented as a package in the common/main.cs module. For your game you will need to use this package or make your own like it. This is in order to gain access to many of the more mundane features of Torque, especially the "administrivia"-like functions that help make your game a finished product but that are not especially exciting in terms of game play features.

Here are the contents of the common/main.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
// Copyright (C) GarageGames.com, Inc.
```


Team LRN

```
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Load up default console values.

Exec("./defaults.cs");

//---------------------------------------------------------------------------

function InitCommon()
{
   // All mods need the random seed set
   SetRandomSeed();

   // Very basic functions used by everyone
   Exec("./client/canvas.cs");
   Exec("./client/audio.cs");
}

function InitBaseClient()
{
   // Base client functionality
   Exec("./client/message.cs");
   Exec("./client/mission.cs");
   Exec("./client/missionDownload.cs");
   Exec("./client/actionMap.cs");
   Exec("./editor/editor.cs");
   Exec("./client/scriptDoc.cs");

   // There are also a number of support scripts loaded by the canvas
   // when it's first initialized.  Check out client/canvas.cs
}

function InitBaseServer()
{
   // Base server functionality
   Exec("./server/audio.cs");
   Exec("./server/server.cs");
   Exec("./server/message.cs");
   Exec("./server/commands.cs");
   Exec("./server/missionInfo.cs");
   Exec("./server/missionLoad.cs");
```

```
   Exec("./server/missionDownload.cs");
   Exec("./server/clientConnection.cs");
   Exec("./server/kickban.cs");
   Exec("./server/game.cs");
}


//--------------------------------------------------------------------------
package Common {

function DisplayHelp() {
   Parent::DisplayHelp();
   Error(
      "Common Mod options:\n"@
      "  -fullscreen             Starts game in full screen mode\n"@
      "  -windowed               Starts game in windowed mode\n"@
      "  -autoVideo              Auto detect video, but prefers OpenGL\n"@
      "  -openGL                 Force OpenGL acceleration\n"@
      "  -directX                Force DirectX acceleration\n"@
      "  -voodoo2                Force Voodoo2 acceleration\n"@
      "  -noSound                Starts game without sound\n"@
      "  -prefs <configFile>    Exec the config file\n"
   );
}

function ParseArgs()
{
   Parent::ParseArgs();

   // Arguments override defaults...
   for (%i = 1; %i < $Game::argc ; %i++)
   {
      %arg = $Game::argv[%i];
      %nextArg = $Game::argv[%i+1];
      %hasNextArg = $Game::argc - %i > 1;

      switch$ (%arg)
      {
         //--------------------
         case "-fullscreen":
            $pref::Video::fullScreen = 1;
            $argUsed[%i]++;
```

```
         //--------------------
         case "-windowed":
            $pref::Video::fullScreen = 0;
            $argUsed[%i]++;

         //--------------------
         case "-noSound":
            error("no support yet");
            $argUsed[%i]++;

         //--------------------
         case "-openGL":
            $pref::Video::displayDevice = "OpenGL";
            $argUsed[%i]++;

         //--------------------
         case "-directX":
            $pref::Video::displayDevice = "D3D";
            $argUsed[%i]++;

         //--------------------
         case "-voodoo2":
            $pref::Video::displayDevice = "Voodoo2";
            $argUsed[%i]++;

         //--------------------
         case "-autoVideo":
            $pref::Video::displayDevice = "";
            $argUsed[%i]++;

         //--------------------
         case "-prefs":
            $argUsed[%i]++;
            if (%hasNextArg) {
               Exec(%nextArg, true, true);
               $argUsed[%i+1]++;
               %i++;
            }
            else
               Error("Error: Missing Command Line argument. Usage: -prefs
<path/script.cs>");
      }
```

```
   }
}

function OnStart()
{
   Parent::OnStart();
   Echo("\n--------- Initializing MOD: Common ---------");
   InitCommon();
}

function OnExit()
{
   Echo("Exporting client prefs");
   Export("$pref::*", "./client/prefs.cs", False);

   Echo("Exporting server prefs");
   Export("$Pref::Server::*", "./server/prefs.cs", False);
   BanList::Export("./server/banlist.cs");

   OpenALShutdown();
   Parent::OnExit();
}

}; // Common package
activatePackage(Common);
```

Two key things that happen during game initialization are calls to `InitBaseServer` and `InitBaseClient`, both of which are defined in common/main.cs. These are critical functions, and yet their actual activities are not that exciting to behold.

```
function InitBaseServer()
{
   Exec("./server/audio.cs");
   Exec("./server/server.cs");
   Exec("./server/message.cs");
   Exec("./server/commands.cs");
   Exec("./server/missionInfo.cs");
   Exec("./server/missionLoad.cs");
   Exec("./server/missionDownload.cs");
   Exec("./server/clientConnection.cs");
   Exec("./server/kickban.cs");
   Exec("./server/game.cs");
}
```

```
function InitBaseClient()
{
   Exec("./client/message.cs");
   Exec("./client/mission.cs");
   Exec("./client/missionDownload.cs");
   Exec("./client/actionMap.cs");
   Exec("./editor/editor.cs");
   Exec("./client/scriptDoc.cs");
}
```

As you can see, both are nothing more than a set of script loading calls. All of the scripts loaded are part of the common code base. We will look at selected key modules from these calls in the rest of this section.

Selected Common Server Modules

Next, we will take a close look at some of the common code server modules. The modules selected are the ones that will best help illuminate how Torque operates.

The Server Module

`InitBaseServer` loads the common server module, server.cs. When we examine this module we see the following functions:

```
PortInit
CreateServer
DestroyServer
ResetServerDefaults
AddToServerGuidList
RemoveFromServerGuidList
OnServerInfoQuery
```

It's not hard to get the sense from that list that this is a pretty critical module!

`PortInit` tries to seize control of the assigned TCP/IP port, and if it can't, it starts incrementing the port number until it finds an open one it can use.

`CreateServer` does the obvious, but it also does some interesting things along the way. First, it makes a call to `DestroyServer!` This is not as wacky as it might seem; while `DestroyServer` does release and disable resources, it does so only after making sure the resources exist. So there's no danger of referencing something that doesn't exist, which would thus cause a crash. You need to specify the server type (single- [default] or multiplayer) and the mission name. The `PortInit` function is called from here, if the server will

be a multiplayer server. The last, but certainly not the least, thing that `CreateServer` does is call `LoadMission`. This call kicks off a long and somewhat involved chain of events that we will cover in a later section.

`DestroyServer` releases and disables resources, as mentioned, and also game mechanisms. It stops further connections from happening and deletes any existing ones; turns off the heartbeat processing; deletes all of the server objects in `MissionGroup`, `MissionCleanup`, and `ServerGroup`; and finally, purges all datablocks from memory.

`ResetServerDefaults` is merely a convenient mechanism for reloading the files in which the server default variable initializations are stored.

`AddToServerGuidList` and `RemoveFromServerGuidList` are two functions for managing the list of clients that are connected to the server.

`OnServerInfoQuery` is a message handler for handling queries from a master server. It merely returns the string `"Doing OK"`. The master server, if there is one, will see this and know that the server is alive. It could say anything—there could even be just a single-space character in the string. The important point is that if the server is *not* doing okay, then the function will not even be called, so the master server would never see the message, would time out, and then would take appropriate action (such as panicking or something useful like that).

The Message Module

`InitBaseServer` loads the common server-side message module, message.cs. Most of this module is dedicated to providing in-game chat capabilities for players.

```
MessageClient
MessageTeam
MessageTeamExcept
MessageAll
MessageAllExcept
ChatMessageClient
ChatMessageTeam
ChatMessageAll
SpamAlert
GameConnection::SpamMessageTimeout
GameConnection::SpamReset
```

The first five functions in the preceding list are for sending server-type messages to individual clients, all clients on a team, and all clients in a game. There are also exception messages where everyone is sent the message *except* a specified client.

Next are the three chat message functions. These are linked to the chat interfaces that players will use to communicate with each other.

These functions all use the `CommandToServer` (see Chapter 6) function internally. It is important to note that there will need to be message handlers for these functions on the client side.

The three spam control functions are used in conjunction with the chat message functions. `SpamAlert` is called just before each outgoing chat message is processed for sending. Its purpose is to detect if a player is swamping the chat window with messages—this action is called *spamming the chat window*. If there are too many messages in a short time frame as determined by the `SpamMessageTimeout` method, then the offending message is suppressed, and an alert message is sent to the client saying something like this: "Enough already! Take a break." Well, you could say it more diplomatically than that, but you get the idea. `SpamReset` merely sets the client's spam state back to normal after an appropriately silent interval.

The MissionLoad Module

Torque has a concept of *mission* that corresponds to what many other games, especially those of the first-person shooter genre, call *maps*. A mission is defined in a mission file that has the extension of .mis. Mission files contain the information that specifies objects in the game world, as well as their placement in the world. Everything that appears in the game world is defined there: items, players, spawn points, triggers, water definitions, sky definitions, and so on.

Missions are downloaded from the server to the client at mission start time or when a client joins a mission already in progress. In this way the server has total control over what the client sees and experiences in the mission.

Here are the contents of the common/server/missionload.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Server mission loading
//-----------------------------------------------------------------------------

// On every mission load except the first, there is a pause after
// the initial mission info is downloaded to the client.
```

```
$MissionLoadPause = 5000;

function LoadMission( %missionName, %isFirstMission )
{
   EndMission();
   Echo("*** LOADING MISSION: " @ %missionName);
   Echo("*** Stage 1 load");

   // Reset all of these
   ClearCenterPrintAll();
   ClearBottomPrintAll();

   // increment the mission sequence (used for ghost sequencing)
   $missionSequence++;
   $missionRunning = false;
   $Server::MissionFile = %missionName;

   // Extract mission info from the mission file,
   // including the display name and stuff to send
   // to the client.
   BuildLoadInfo( %missionName );

   // Download mission info to the clients
   %count = ClientGroup.GetCount();
   for( %cl = 0; %cl < %count; %cl++ ) {
      %client = ClientGroup.GetObject( %cl );
      if (!%client.IsAIControlled())
         SendLoadInfoToClient(%client);
   }

   // if this isn't the first mission, allow some time for the server
   // to transmit information to the clients:
   if( %isFirstMission || $Server::ServerType $= "SinglePlayer" )
      LoadMissionStage2();
   else
      schedule( $MissionLoadPause, ServerGroup, LoadMissionStage2 );
}

function LoadMissionStage2()
{
   // Create the mission group off the ServerGroup
   Echo("*** Stage 2 load");
```

```
$instantGroup = ServerGroup;

// Make sure the mission exists
%file = $Server::MissionFile;

if( !IsFile( %file ) ) {
   Error( "Could not find mission " @ %file );
   return;
}

// Calculate the mission CRC.  The CRC is used by the clients
// to cache mission lighting.
$missionCRC = GetFileCRC( %file );

// Exec the mission, objects are added to the ServerGroup
Exec(%file);

// If there was a problem with the load, let's try another mission
if( !IsObject(MissionGroup) ) {
   Error( "No 'MissionGroup' found in mission \"" @ $missionName @ "\"." );
   schedule( 3000, ServerGroup, CycleMissions );
   return;
}

// Mission cleanup group
new SimGroup( MissionCleanup );
$instantGroup = MissionCleanup;

// Construct MOD paths
PathOnMissionLoadDone();

// Mission loading done...
Echo("*** Mission loaded");

// Start all the clients in the mission
$missionRunning = true;
for( %clientIndex = 0; %clientIndex < ClientGroup.GetCount(); %clientIndex++ )
   ClientGroup.GetObject(%clientIndex).LoadMission();

// Go ahead and launch the game
OnMissionLoaded();
PurgeResources();
```

Team LRN

```
}

function EndMission()
{
   if (!IsObject( MissionGroup ))
      return;

   Echo("*** ENDING MISSION");

   // Inform the game code we're done.
   OnMissionEnded();

   // Inform the clients
   for( %clientIndex = 0; %clientIndex < ClientGroup.GetCount(); %clientIndex++ ) {
      // clear ghosts and paths from all clients
      %cl = ClientGroup.GetObject( %clientIndex );
      %cl.EndMission();
      %cl.ResetGhosting();
      %cl.ClearPaths();
   }

   // Delete everything
   MissionGroup.Delete();
   MissionCleanup.Delete();

   $ServerGroup.Delete();
   $ServerGroup = new SimGroup(ServerGroup);
}

function ResetMission()
{
   Echo("*** MISSION RESET");

   // Remove any temporary mission objects
   MissionCleanup.Delete();
   $instantGroup = ServerGroup;
   new SimGroup( MissionCleanup );
   $instantGroup = MissionCleanup;

   //
   OnMissionReset();
}
```

Here are the mission loading–oriented functions on the server contained in this module:

```
LoadMission
LoadMissionStage2
EndMission
ResetMission
```

`LoadMission`, as we saw in an earlier section, is called in the `CreateServer` function. It kicks off the process of loading a mission onto the server. Mission information is assembled from the mission file and sent to all the clients for display to their users.

After the mission file loads, `LoadMissionStage2` is called. In this function, the server calculates the CRC value for the mission and saves it for later use.

Once the mission is successfully loaded onto the server, each client is sent the mission via a call to its `GameConnection` object's `LoadMission` method.

What's a CRC Value, and Why Should I Care?

We use a *Cyclic Redundancy Check* (CRC) when transmitting data over potentially error-prone media. Networking protocols use CRCs at a low level to verify that the sent data is the same data that was received.

A CRC is a mathematical computation performed on data that arrives at a number that represents both the content of the data and how it's arranged. The point is that the number, called a *checksum*, uniquely identifies the set of data, like a fingerprint.

By comparing the checksum of a set of data to another data set's checksum, you can decide if the two data sets are identical.

Why should you care? Well, in addition to the simple goal of maintaining data integrity, CRCs are another arrow in your anticheat quiver. You can use CRCs to ensure that files stored on the clients are the same as the files on the server and, in this regard, that all the clients have the same files—the result is that the playing field is level.

`EndMission` releases resources and disables other mission-related mechanisms, clearing the server to load a new mission when tasked to do so.

`ResetMission` can be called from the `EndGame` function in the control/server/misc/game.cs module to prepare the server for a new mission if you are using mission cycling techniques.

The MissionDownload Module

Here are the contents of the common/server/missiondownload.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Mission Loading
// The server portion of the client/server mission loading process
//-----------------------------------------------------------------------------

function GameConnection::LoadMission(%this)
{
   // Send over the information that will display the server info.
   // When we learn it got there, we'll send the datablocks.
   %this.currentPhase = 0;
   if (%this.IsAIControlled())
   {
      // Cut to the chase...
      %this.OnClientEnterGame();
   }
   else
   {
      CommandToClient(%this, 'MissionStartPhase1', $missionSequence,
         $Server::MissionFile, MissionGroup.musicTrack);
      Echo("*** Sending mission load to client: " @ $Server::MissionFile);
   }
}

function ServerCmdMissionStartPhase1Ack(%client, %seq)
{
   // Make sure to ignore calls from a previous mission load
   if (%seq != $missionSequence || !$MissionRunning)
      return;
   if (%client.currentPhase != 0)
      return;
   %client.currentPhase = 1;

   // Start with the CRC
   %client.SetMissionCRC( $missionCRC );
```

```
   // Send over the datablocks...
   // OnDataBlocksDone will get called when have confirmation
   // that they've all been received.
   %client.TransmitDataBlocks($missionSequence);
}

function GameConnection::OnDataBlocksDone( %this, %missionSequence )
{
   // Make sure to ignore calls from a previous mission load
   if (%missionSequence != $missionSequence)
      return;
   if (%this.currentPhase != 1)
      return;
   %this.currentPhase = 1.5;

   // On to the next phase
   CommandToClient(%this, 'MissionStartPhase2', $missionSequence, $Server::MissionFile);
}

function ServerCmdMissionStartPhase2Ack(%client, %seq)
{
   // Make sure to ignore calls from a previous mission load
   if (%seq != $missionSequence || !$MissionRunning)
      return;
   if (%client.currentPhase != 1.5)
      return;
   %client.currentPhase = 2;

   // Update mod paths, this needs to get there before the objects.
   %client.TransmitPaths();

   // Start ghosting objects to the client
   %client.ActivateGhosting();

}

function GameConnection::ClientWantsGhostAlwaysRetry(%client)
{
   if($MissionRunning)
      %client.ActivateGhosting();
}
```

```
function GameConnection::OnGhostAlwaysFailed(%client)
{

}

function GameConnection::OnGhostAlwaysObjectsReceived(%client)
{
   // Ready for next phase.
   CommandToClient(%client, 'MissionStartPhase3', $missionSequence, $Server::Mission-
File);
}

function ServerCmdMissionStartPhase3Ack(%client, %seq)
{
   // Make sure to ignore calls from a previous mission load
   if(%seq != $missionSequence || !$MissionRunning)
      return;
   if(%client.currentPhase != 2)
      return;
   %client.currentPhase = 3;

   // Server is ready to drop into the game
   %client.StartMission();
   %client.OnClientEnterGame();
}
```

The following functions and `GameConnection` methods are defined in the MissionDownload module:

```
GameConnection::LoadMission
GameConnection::OnDataBlocksDone
GameConnection::ClientWantsGhostAlwaysRetry
GameConnection::OnGhostAlwaysFailed
GameConnection::OnGhostAlwaysObjectsReceived
ServerCmdMissionStartPhase1Ack
ServerCmdMissionStartPhase2Ack
ServerCmdMissionStartPhase3Ack
```

This module handles the server-side activities in the mission download process (see Figure 7.1).

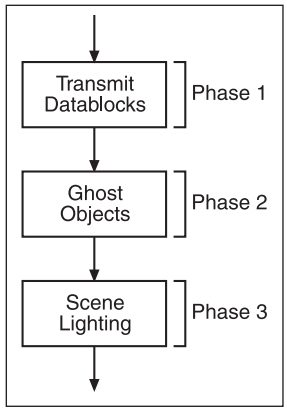


Figure 7.1 Mission download phases.

This module contains the mission download methods for each client's `GameConnection` object.

The download process for the client object starts when its `LoadMission` method in this module is called at the end of the server's `LoadMissionStage2` function in the server's MissionLoad module described in the previous section. It then embarks on a phased series of activities coordinated between the client server (see Figure 7.2). The messaging system for this process is the `CommandToServer` and `CommandToClient` pair of direct messaging functions.

The server invokes the client `MissionStartPhase`*n* (where *n* is 1, 2, or 3) function to request permission to start each phase. This is done using our old friend `CommandToServer`. When a client is ready for a phase, it responds with a `MissionStartPhase`*n*`Ack` message, for which there is a handler on the server contained in this module.

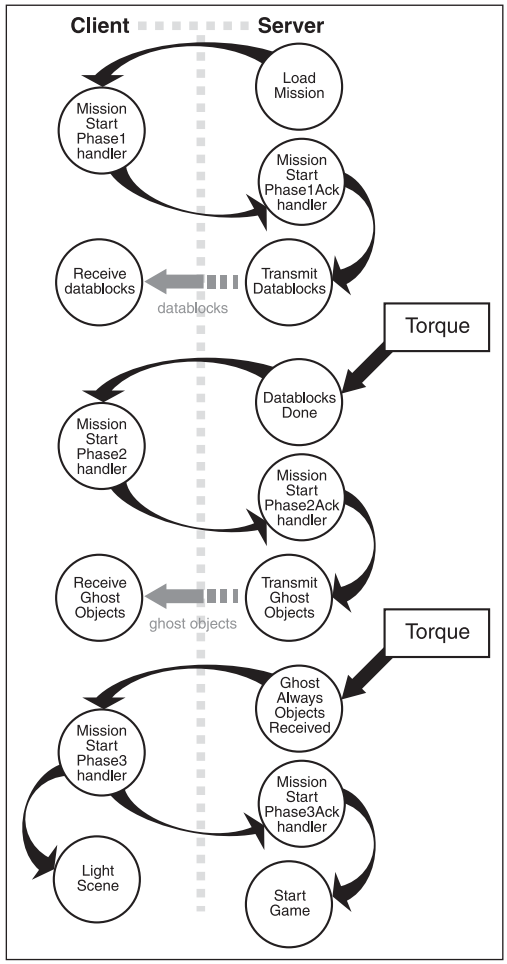


Figure 7.2 Mission download process.

The handler `GameConnection::onDataBlocksDone` is invoked when phase one has finished. This handler then initiates phase two by sending the `MissionStartPhase2` message to the client.

The `GameConnection::onGhostAlwaysObjects Received` handler is invoked when phase two is completed. At the end of this phase, the client has all of the data needed to replicate the server's version of any dynamic objects in the game that are ghosted to the clients. This handler then sends the `MissionStartPhase3` message to the client.

When the server receives the `MissionStart-Phase3Ack` message, it then starts the mission for each client, inserting the client into the game.

The ClientConnection Module

The ClientConnection module is where most of the server-side code for dealing with clients is located. Here are the contents of the common/server/clientconnection.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//-----------------------------------------------------------------------------

function GameConnection::OnConnectRequest( %client, %netAddress, %name )
{
   Echo("Connect request from: " @ %netAddress);
   if($Server::PlayerCount >= $pref::Server::MaxPlayers)
      return "CR_SERVERFULL";
   return "";
}

function GameConnection::OnConnect( %client, %name )
{
   MessageClient(%client,'MsgConnectionError',"",$Pref::Server::ConnectionError);

   SendLoadInfoToClient( %client );

   if (%client.getAddress() $= "local") {
      %client.isAdmin = true;
      %client.isSuperAdmin = true;
   }

   %client.guid = 0;
   AddToServerGuidList( %client.guid );

   // Set admin status
   %client.isAdmin = false;
   %client.isSuperAdmin = false;

   // Save client preferences on the Connection object for later use.
   %client.gender = "Male";
   %client.armor = "Light";
   %client.race = "Human";
   %client.skin = AddTaggedString( "base" );
   %client.SetPlayerName(%name);
   %client.score = 0;

   $instantGroup = MissionCleanup;
```

```
Echo("CADD: " @ %client @ " " @ %client.GetAddress());

// Inform the client of all the other clients
%count = ClientGroup.GetCount();
for (%cl = 0; %cl < %count; %cl++) {
   %other = ClientGroup.GetObject(%cl);
   if ((%other != %client)) {

      MessageClient(%client, 'MsgClientJoin', "",
            %other.name,
            %other,
            %other.sendGuid,
            %other.score,
            %other.IsAIControlled(),
            %other.isAdmin,
            %other.isSuperAdmin);
   }
}

// Inform the client we've joined up
MessageClient(%client,
   'MsgClientJoin', '\c2Welcome to the Torque demo app %1.',
   %client.name,
   %client,
   %client.sendGuid,
   %client.score,
   %client.IsAiControlled(),
   %client.isAdmin,
   %client.isSuperAdmin);

// Inform all the other clients of the new guy
MessageAllExcept(%client, -1, 'MsgClientJoin', '\c1%1 joined the game.',
   %client.name,
   %client,
   %client.sendGuid,
   %client.score,
   %client.IsAiControlled(),
   %client.isAdmin,
   %client.isSuperAdmin);

// If the mission is running, go ahead and download it to the client
if ($missionRunning)
```

Team LRN

```
      %client.LoadMission();
   $Server::PlayerCount++;
}

function GameConnection::SetPlayerName(%client,%name)
{
   %client.SendGuid = 0;

   // Minimum length requirements
   %name = StripTrailingSpaces( StrToPlayerName( %name ) );
   if ( Strlen( %name ) < 3 )
      %name = "Poser";

   // Make sure the alias is unique, we'll hit something eventually
   if (!IsNameUnique(%name))
   {
      %isUnique = false;
      for (%suffix = 1; !%isUnique; %suffix++)  {
         %nameTry = %name @ "." @ %suffix;
         %isUnique = IsNameUnique(%nameTry);
      }
      %name = %nameTry;
   }
   // Tag the name with the "smurf" color:
   %client.nameBase = %name;
   %client.name = AddTaggedString("\cp\c8" @ %name @ "\co");
}

function IsNameUnique(%name)
{
   %count = ClientGroup.GetCount();
   for ( %i = 0; %i < %count; %i++ )
   {
      %test = ClientGroup.GetObject( %i );
      %rawName = StripChars( detag( GetTaggedString( %test.name ) ),
"\cp\co\c6\c7\c8\c9" );
      if ( Strcmp( %name, %rawName ) == 0 )
         return false;
   }
   return true;
}
```

```
function GameConnection::OnDrop(%client, %reason)
{
   %client.OnClientLeaveGame();

   RemoveFromServerGuidList( %client.guid );
   MessageAllExcept(%client, -1, 'MsgClientDrop', '\c1%1 has left the game.',
%client.name, %client);

   RemoveTaggedString(%client.name);
   Echo("CDROP: " @ %client @ " " @ %client.GetAddress());
   $Server::PlayerCount--;

   if( $Server::PlayerCount == 0 && $Server::Dedicated)
      Schedule(0, 0, "ResetServerDefaults");
}

function GameConnection::StartMission(%this)
{
   CommandToClient(%this, 'MissionStart', $missionSequence);
}

function GameConnection::EndMission(%this)
{
   CommandToClient(%this, 'MissionEnd', $missionSequence);
}

function GameConnection::SyncClock(%client, %time)
{
   CommandToClient(%client, 'syncClock', %time);
}

function GameConnection::IncScore(%this,%delta)
{
   %this.score += %delta;
   MessageAll('MsgClientScoreChanged', "", %this.score, %this);
}
```

The following functions and `GameConnection` methods are defined in the ClientConnection module:

```
GameConnection::OnConnectRequest
GameConnection::OnConnect
GameConnection::SetPlayerName
```

```
IsNameUnique
GameConnection::OnDrop
GameConnection::StartMission
GameConnection::EndMission
GameConnection::SyncClock
GameConnection::IncScore
```

The method `GameConnection::OnConnectRequest` is the server-side destination of the client-side `GameConnection::Connect` method. We use this method to vet the request—for example, examine the IP address to compare to a ban list, or make sure that the server is not full, and stuff like that. We have to make sure that if we want to allow the request, we must return a null string ("").

The next method, `GameConnection::OnConnect`, is called after the server has approved the connection request. We get a client handle and a name string passed in as parameters. The first thing we do is ship down to the client a tagged string to indicate that a connection error has happened. We do not tell the client to use this string. It's just a form of preloading the client.

Then we send the load information to the client. This is the mission information that the client can display to the user while the mission loading process takes place. After that, if the client also happens to be the host (entirely possible), we set the client to be a superAdmin.

Then we add the client to the user ID list that the server maintains. After that there are a slew of game play client settings we can initialize.

Next, we start a series of notifications. First, we tell all clients that the player has joined the server. Then we tell the joining player that he is indeed welcome here, despite possible rumors to the contrary. Finally, we tell all the client-players that there is a new kid on the block, so go kill him. Or some such—whatever you feel like!

After all the glad-handing is done, we start downloading the mission data to the client starting the chain of events depicted back there in Figure 7.2.

`GameConnection::SetPlayerName` does some interesting name manipulation. First, it tidies up any messy names that have leading or trailing spaces. We don't like names that are too short (trying to hide something?), so we don't allow those names. Then we make sure that the name is not already in use. If it is, then an instance number is added to the end of the name. The name is converted to a tagged string so that the full name only gets transmitted once to each client; then the tag number is used after that, if necessary.

The function `IsNameUnique` searches through the server's name list looking for a match. If it finds the name, then it isn't unique; otherwise it is.

Team LRN

The method `GameConnection::OnDrop` is called when the decision is made to drop a client. First, the method makes a call to the client so that it knows how to act during the drop. Then it removes the client from its internal list. All clients (except the one dropped) are sent a server text message notifying them of the drop, which they can display. After the last player leaves the game, this method restarts the server. For a persistent game, this statement should probably be removed.

The next method, `GameConnection::StartMission`, simply notifies clients whenever the server receives a command to start another server session in order to give the clients time to prepare for the near-future availability of the server. The `$missionSequence` is used to manage mission ordering, if needed.

Next, `GameConnection::EndMission` is used to notify clients that a mission is ended, and hey! Stop playing already!

The method `GameConnection::SyncClock` is used to make sure that all clients' timers are synchronized with the server. You can call this function for a client anytime after the mission is loaded, but before the client's player has spawned.

Finally, the method `GameConnection::IncScore` is called whenever you want to reward a player for doing well. By default, this method is called when a player gets a kill on another player. When the player's score is incremented, all other players are notified, via their clients, of the score.

The Game Module

The server-side Game module is the logical place to put server-specific game play features. Here are the contents of the common/server/game.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//-----------------------------------------------------------------------------
function OnServerCreated()
{
   $Server::GameType = "Test App";
   $Server::MissionType = "Deathmatch";
}

function OnServerDestroyed()
{
   DestroyGame();
}
```

```
function OnMissionLoaded()
{
   StartGame();
}

function OnMissionEnded()
{
   EndGame();
}

function OnMissionReset()
{
   // stub
}

function GameConnection::OnClientEnterGame(%this)
{
//stub
}

function GameConnection::OnClientLeaveGame(%this)
{
//stub
}

//---------------------------------------------------------------------------
// Functions that implement game-play
//---------------------------------------------------------------------------
function StartGame()
{
//stub
}

function EndGame()
{
//stub
}
```

The following functions and GameConnection methods are defined in the Game module:

- `OnServerCreated`
- `OnServerDestroyed`
- `OnMissionLoaded`

```
OnMissionEnded
OnMissionReset
StartGame
EndGame
GameConnection::OnClientEnterGame
GameConnection::OnClientLeaveGame
```

The first function defined, `OnServerCreated`, is called from `CreateServer` when a server is constructed. It is a useful place to load server-specific datablocks.

The variable `$Server::GameType` is sent to the master, if one is used. Its purpose is to uniquely identify the game and distinguish it from other games handled by the master server. The variable `$Server::MissionType` is also sent to the server—clients can use its value to filter servers based on mission type.

The next function, `OnServerDestroyed`, is the antithesis of `OnServerCreated`—anything you do there should be undone in this function.

The function `OnMissionLoaded` is called by `LoadMission` once a mission has finished loading. This is a great location to initialize mission-based game play features, like perhaps calculating weather effects based on a rotating mission scheme.

`OnMissionEnded` is called by `EndMission` just before it is destroyed; this is where you should undo anything you did in `OnMissionLoaded`.

`OnMissionReset` is called by `ResetMission`, after all the temporary mission objects have been deleted.

The function `GameConnection::OnClientEnterGame` is called for each client after it has finished downloading the mission and is ready to start playing. This would be a good place to load client-specific persistent data from a database back end, for example.

`GameConnection::OnClientLeaveGame` is called for each client that is dropped. This would be a good place to do a final update of back-end database information for the client.

Although we don't use a great deal of the functions in this module, it is a great location for a lot of game play features to reside.

Selected Common Code Client Modules

Next, we will take a close look at some of the common code client modules. The modules selected are the ones that will best help illuminate how Torque operates.

Keep in mind that all of these modules are designed to affect things that concern the local client, even though they might require contacting the server from time to time.

Team LRN

This point is important: When you add features or capabilities, you must always keep in mind whether you want the feature to affect only the local client (such as some user preference change) or you want the feature to affect all clients. In the latter case it would be best to use modules that are server-resident when they run.

The Canvas Module

The Canvas module is another one of those simple, small, but critical modules. One of the key features of this module is that the primary function contained in here, `InitCanvas`, loads a number of general graphical user interface support modules. This module is loaded from the `InitCommon` function, rather than from the `InitBaseClient` function, which is where the rest of the key common modules get loaded. Here are the contents of the common/client/canvas.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Function to construct and initialize the default canvas window
// used by the games

function InitCanvas(%windowName)
{
   VideoSetGammaCorrection($pref::OpenGL::gammaCorrection);
   if (!CreateCanvas(%windowName)) {
      Quit();
      return;
   }

   SetOpenGLTextureCompressionHint( $pref::OpenGL::compressionHint );
   SetOpenGLAnisotropy( $pref::OpenGL::anisotropy );
   SetOpenGLMipReduction( $pref::OpenGL::mipReduction );
   SetOpenGLInteriorMipReduction( $pref::OpenGL::interiorMipReduction );
   SetOpenGLSkyMipReduction( $pref::OpenGL::skyMipReduction );

   // Declare default GUI Profiles.
   Exec("~/ui/defaultProfiles.cs");

   // Common GUI's
   Exec("~/ui/GuiEditorGui.gui");
```

```
   Exec("~/ui/ConsoleDlg.gui");
   Exec("~/ui/InspectDlg.gui");
   Exec("~/ui/LoadFileDlg.gui");
   Exec("~/ui/SaveFileDlg.gui");
   Exec("~/ui/MessageBoxOkDlg.gui");
   Exec("~/ui/MessageBoxYesNoDlg.gui");
   Exec("~/ui/MessageBoxOKCancelDlg.gui");
   Exec("~/ui/MessagePopupDlg.gui");
   Exec("~/ui/HelpDlg.gui");
   Exec("~/ui/RecordingsDlg.gui");

   // Commonly used helper scripts
   Exec("./metrics.cs");
   Exec("./messageBox.cs");
   Exec("./screenshot.cs");
   Exec("./cursor.cs");
   Exec("./help.cs");
}

function ResetCanvas()
{
   if (IsObject(Canvas))
   {
      Canvas.Repaint();
   }
}
```

`InitCanvas` is obviously the main function in this module. When it is called, it first calls `VideoSetGammaCorrection` using a global preferences variable. If the value passed is 0 or undefined, then there is no change in the gamma correction (see Table 7.1).

Then we attempt to create the canvas, which is an abstracted call to the Windows API to create a window. The `%windowName` variable is passed in as a string that sets the window's title. If we can't create the window, we quit because there is no point continuing without any means to display our game.

Following that, there is a series of OpenGL settings, again using global preference variables. See Table 7.1 for an explanation of these settings.

Table 7.1 OpenGL Settings

Module	Function
GammaCorrection	Gamma correction modifies the overall brightness of an image. Images that are not corrected can look either overbleached or too dark.

TextureCompressionHint	The choice of how much texture compression (to reduce memory and graphics transfer bandwidth) to employ is left up to the drivers and hardware, but we can hint at how we would like the compression to work, if feasible. Valid hints are:
	`GL_DONT_CARE`
	`GL_FASTEST`
	`GL_NICEST`

Anisotropy	Anisotropic filtering is used to address a specific kind of texture artifact that occurs when a 3D surface is sloped relative to the view camera. The higher the value set for this (between 0 and 1, exclusive), the more filtering is performed by the hardware. Too high a setting might cause too much fuzziness in an image.
MipReduction	See Chapter 3 for a discussion of mipmapping. This value can be from 0 to 5. The higher the number, the more mipmapping levels supported. Image textures must be created to support these levels in order to achieve the best effect.
InteriorMipReduction	The same as MipReduction, but for use in interiors (.dif file format models).
SkyMipReduction	The same as MipReduction, but for use in skybox images.

Next, the function loads a bunch of support files that establish user interface mechanisms, dialogs, and profiles for describing them.

Then there is a series of calls to load modules that provide access to some common utility functions that can be used for measuring performance, taking screen shots, displaying Help information, and so on.

The `ResetCanvas` function checks to see if a canvas object exists, and if so, `ResetCanvas` then forces it to be repainted (re-rendered).

The Mission Module

The Mission module doesn't really do much. Its existence is no doubt because some forethought had been given to future expansion directions for the common code scripts. Here are the contents of the common/client/mission.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
```

```
// Copyright (c) 2001 GarageGames.com
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Mission start / end events sent from the server
//-----------------------------------------------------------------------------

function ClientCmdMissionStart(%seq)
{
   // The client receives a mission start right before
   // being dropped into the game.
}

function ClientCmdMissionEnd(%seq)
{
   // Disable mission lighting if it's going; this is here
   // in case the mission ends while we are in the process
   // of loading it.
   $lightingMission = false;
   $sceneLighting::terminateLighting = true;
}
```

`ClientCmdMissionStart` is a stub routine—not much to say here other than this routine gets called immediately before the client-player finds himself in the game. This is a handy place for last-minute client-side code—the mission is known and loaded, and all objects are ghosted, including any remote clients. This might be a good place to build and display a map or to possibly fire up an Internet Relay Chat session, if you have written one for yourself in Torque Script (it is possible—a member of the GarageGames community has done just that).

`ClientCmdMissionEnd` resets some lighting variables. This would be the place to undo anything you started in the `ClientCmdMissionStart` function.

The thing that makes this module, and therefore its functions, key is its existence. You should consider utilizing these functions in your game and expanding their functionality.

The MissionDownload Module

Just as the server side has a module called MissionDownload, so has the client code. It certainly can be confusing, so you have to stay on your toes when dealing with these modules, always being aware of whether you are dealing with the client or the server version. The choice of names is understandable though, when you realize that they are functionally complementary—the mission download activity required synchronized and coordinated actions from both the client and the server. Two peas in a pod.

Team LRN

Here are the contents of the common/client/missiondownload.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Phase 1
//-----------------------------------------------------------------------------

function ClientCmdMissionStartPhase1(%seq, %missionName, %musicTrack)
{
   // These need to come after the cls.
   Echo ("*** New Mission: " @ %missionName);
   Echo ("*** Phase 1: Download Datablocks & Targets");
   OnMissionDownloadPhase1(%missionName, %musicTrack);
   CommandToServer('MissionStartPhase1Ack', %seq);
}

function OnDataBlockObjectReceived(%index, %total)
{
   OnPhase1Progress(%index / %total);
}

//-----------------------------------------------------------------------------
// Phase 2
//-----------------------------------------------------------------------------

function ClientCmdMissionStartPhase2(%seq,%missionName)
{
   onPhase1Complete();
   Echo ("*** Phase 2: Download Ghost Objects");
   purgeResources();
   onMissionDownloadPhase2(%missionName);
   commandToServer('MissionStartPhase2Ack', %seq);
}

function OnGhostAlwaysStarted(%ghostCount)
{
   $ghostCount = %ghostCount;
   $ghostsRecvd = 0;
```

```
}

function OnGhostAlwaysObjectReceived()
{
   $ghostsRecvd++;
   OnPhase2Progress($ghostsRecvd / $ghostCount);
}

//--------------------------------------------------------------------------
// Phase 3
//--------------------------------------------------------------------------

function ClientCmdMissionStartPhase3(%seq,%missionName)
{
   OnPhase2Complete();
   StartClientReplication();
   StartFoliageReplication();
   Echo ("*** Phase 3: Mission Lighting");
   $MSeq = %seq;
   $Client::MissionFile = %missionName;

   // Need to light the mission before we are ready.
   // The sceneLightingComplete function will complete the handshake
   // once the scene lighting is done.
   if (LightScene("SceneLightingComplete", ""))
   {
      Error("Lighting mission....");
      schedule(1, 0, "UpdateLightingProgress");
      OnMissionDownloadPhase3(%missionName);
      $lightingMission = true;
   }
}

function UpdateLightingProgress()
{
   OnPhase3Progress($SceneLighting::lightingProgress);
   if ($lightingMission)
      $lightingProgressThread = schedule(1, 0, "UpdateLightingProgress");
}

function SceneLightingComplete()
{
```

```
   Echo("Mission lighting done");
   OnPhase3Complete();

   // The is also the end of the mission load cycle.
   OnMissionDownloadComplete();
   CommandToServer('MissionStartPhase3Ack', $MSeq);
}
```

When reviewing this module, you should refer back to the server-side MissionDownload module descriptions and Figures 7.1 and 7.2.

The first function for phase one, `ClientCmdMissionStartPhase1`, calls the function `OnMissionDownloadPhase1`, which is something you want to define in your control code. Its basic purpose is to set up for a progress display as the datablocks are loaded. As soon as this call returns, an acknowledgment is sent back to the server using `CommandToServer` to send the `MissionStartPhase1Ack` message back. At this time it also reflects the sequence number (`%seq`) back to the server, to ensure that the client and server remain synchronized.

The next function, `OnDataBlockObjectReceived`, is an important one. This message handler gets called every time the Torque Engine client-side code detects that it has finished receiving a datablock. When invoked, it then calls `onPhase1Progress`, which needs to be defined in our control client code.

The next function, `ClientCmdMissionStartPhase2`, is part of the phase two activities. Its duties are much the same as for `ClientCmdMissionStartPhase1`, but this time using `OnMissionDownloadPhase2` and `MissionStartPhase2Ack`.

The next function, `OnGhostAlwaysStarted`, is called by the engine after it processes the `MissionStartPhase2Ack` message. It is used to track ghosted object counts.

When an object has been successfully ghosted, `onGhostAlwaysObjectReceived` is called from the engine. We use this to call `onPhase2Progress` in order to update our progress display.

The `ClientCmdMissionStartPhase3` function is the last in the series. When it is called, we update our progress display and then turn on two client-side replication functions. These functions provide special objects (such as grass and trees) that will be computed and rendered only by the client. For example, the server sends a seed for the location of a tuft of grass. The client-side replication code calculates the locations of hundreds or even thousands of copies of this tuft of grass and distributes them appropriately.

Because these objects are deemed not to be critical for game play, we can take the risk of client-side computation without risking someone modifying the code to cheat. Someone could modify the code, but it wouldn't gain him any online advantage.

Next we call the function `LightScene` to perform the scene's terrain and interior lighting passes. We pass the completion callback function `SceneLightingComplete`, which will be called when the lighting calculations are finished.

Team LRN

We also schedule a function (`UpdateLightingProgress`) to be repeatedly called while the lighting is under way, as follows:

```
schedule(1, 0, "updateLightingProgress");
```

In this case the function is called after one millisecond.

`UpdateLightingProgress` is a short function. It makes a call to update the progress display and then schedules itself to be called again in another millisecond if the lighting is not finished. It can tell if the lighting is finished by checking the variable `$lightingMission`. If it is `true`, then lighting is still under way.

`SceneLightingComplete` is the completion callback passed to `LightScene`. When `SceneLighting-Complete` is called, lighting has completed, so it sets the variable `$lightingMission` to `false`, which will eventually, within a millisecond or so, be detected by `UpdateLightingProgress`. It then notifies the server that lighting is complete by sending the `MissionStartPhase3Ack` message. And away we go!

The Message Module

The Message module provides front-end generic message handlers for two defined message types, as well as a tool for installing handlers at run time. You may or may not find this useful, but a look at how these functions work will help when it comes to creating your own sophisticated messaging system. Here are the contents of the common/client/message.cs module.

```
//-----------------------------------------------------------------------------
// Torque Game Engine
//
// Copyright (c) 2001 GarageGames.com
// Portions Copyright (c) 2001 by Sierra Online, Inc.
//-----------------------------------------------------------------------------

function ClientCmdChatMessage(%sender, %voice, %pitch, %msgString, %a1, %a2, %a3, %a4,
%a5, %a6, %a7, %a8, %a9, %a10)
{
   OnChatMessage(detag(%msgString), %voice, %pitch);
}

function ClientCmdServerMessage(%msgType, %msgString, %a1, %a2, %a3, %a4, %a5, %a6, %a7,
%a8, %a9, %a10)
{
   // Get the message type; terminates at any whitespace.
   %tag = GetWord(%msgType, 0);
```

```
   // First see if there is a callback installed that doesn't have a type;
   // if so, that callback is always executed when a message arrives.
   for (%i = 0; (%func = $MSGCB["", %i]) !$= ""; %i++) {
      call(%func, %msgType, %msgString, %a1, %a2, %a3, %a4, %a5, %a6, %a7, %a8, %a9,
%a10);
   }

   // Next look for a callback for this particular type of ServerMessage.
   if (%tag !$= "") {
      for (%i = 0; (%func = $MSGCB[%tag, %i]) !$= ""; %i++) {
         call(%func, %msgType, %msgString, %a1, %a2, %a3, %a4, %a5, %a6, %a7, %a8, %a9,
%a10);
      }
   }
}

function AddMessageCallback(%msgType, %func)
{
   for (%i = 0; (%afunc = $MSGCB[%msgType, %i]) !$= ""; %i++) {
      // If it already exists as a callback for this type,
      // nothing to do.
      if (%afunc $= %func) {
         return;
      }
   }
   // Set it up.
   $MSGCB[%msgType, %i] = %func;
}

function DefaultMessageCallback(%msgType, %msgString, %a1, %a2, %a3, %a4, %a5, %a6, %a7,
%a8, %a9, %a10)
{
   OnServerMessage(detag(%msgString));
}

AddMessageCallback("", DefaultMessageCallback);
```

The first function, `ClientCmdChatMessage`, is for chat messages only and is invoked on the client when the server uses the `CommandToClient` function with the message type `ChatMessage`. Refer back to the server-side message module if you need to. The first parameter (`%sender`) is the `GameConnection` object handle of the player that sent the chat message. The second parameter (`%voice`) is an Audio Voice identifier string. Parameter three (`%pitch`) will also be covered in the audio chapter later. Finally, the fourth parameter (`%msgString`) contains the

actual chat message in a tagged string. The rest of the parameters are not actually acted on so can be safely ignored for now. The parameters are passed on to the pseudo-handler `OnChatMessage`. It's called a *pseudo-handler* because the function that calls `OnChatMessage` is not really calling out from the engine. However, it is useful to treat this operation as if a callback message and handler were involved for conceptual reasons.

The next function, `ClientCmdServerMessage`, is used to deal with game event descriptions, which may or may not include text messages. These can be sent using the message functions in the server-side Message module. Those functions use `CommandToClient` with the type `ServerMessage`, which invokes the function described next.

For `ServerMessage` messages, the client can install callbacks that will be run according to the type of the message.

Obviously, `ClientCmdServerMessage` is more involved. After it uses the `GetWord` function to extract the message type as the first text word from the string `%msgType`, it iterates through the message callback array (`$MSGCB`) looking for any untyped callback functions and executes them all. It then goes through the array again, looking for registered callback functions with the same message type as the incoming message, executing any that it finds.

The next function, `addMessageCallback`, is used to register callback functions in the `$MSGCB` message callback array. This is not complex; `addMessageCallback` merely steps through the array looking for the function to be registered. If it isn't there, `addMessageCallback` stores a handle to the function in the next available slot.

The last function, `DefaultMessageCallback`, is supplied in order to provide an untyped message to be registered. The registration takes place with the line after the function definition.

A Final Word

The common code base includes a ton of functions and methods. We have only touched on about half of them here. I aimed to show you the most important modules and their contents, and I think that's been accomplished nicely. For your browsing pleasure, Table 7.2 contains a reference to find all the functions in all common code modules.

Table 7.2 Common Code Functions

Module	Function
common/main.cs	InitCommon
	InitBaseClient
	InitBaseServer
	DisplayHelp
	ParseArgs
	OnStart
	OnExit
common/client/actionMap.cs	ActionMap::copyBind
	ActionMap::blockBind
common/client/audio.cs	OpenALInit
	OpenALShutdown
common/client/canvas.cs	InitCanvas
	ResetCanvas
common/client/cursor.cs	CursorOff
	CursorOn
	GuiCanvas::checkCursor
	GuiCanvas::setContent
	GuiCanvas::pushDialog
	GuiCanvas::popDialog
	GuiCanvas::popLayer
common/client/help.cs	HelpDlg::onWake
	HelpFileList::onSelect
	GetHelp
	ContextHelp
	GuiControl::getHelpPage
	GuiMLTextCtrl::onURL
common/client/message.cs	ClientCmdChatMessage
	ClientCmdServerMessage
	AddMessageCallback
	DefaultMessageCallback
common/client/messageBox.cs	MessageCallback
	MBSetText
	MessageBoxOK
	MessageBoxOKDlg::onSleep
	MessageBoxOKCancel
	MessageBoxOKCancelDlg::onSleep
	MessageBoxYesNo
	MessageBoxYesNoDlg::onSleep
	MessagePopup
	CloseMessagePopup

continued

common/client/metrics.cs	FpsMetricsCallback
	TerrainMetricsCallback
	VideoMetricsCallback
	InteriorMetricsCallback
	TextureMetricsCallback
	WaterMetricsCallback
	TimeMetricsCallback
	VehicleMetricsCallback
	AudioMetricsCallback
	DebugMetricsCallback
	Metrics
common/client/mission.cs	ClientCmdMissionStart
	ClientCmdMissionEnd
common/client/missionDownload.cs	ClientCmdMissionStartPhase1
	OnDataBlockObjectReceived
	ClientCmdMissionStartPhase2
	OnGhostAlwaysStarted
	OnGhostAlwaysObjectReceived
	ClientCmdMissionStartPhase3
	UpdateLightingProgress
	SceneLightingComplete
common/client/recordings.cs	RecordingsDlg::onWake
	StartSelectedDemo
	StartDemoRecord
	StopDemoRecord
	DemoPlaybackComplete
common/client/screenshot.cs	FormatImageNumber
	RecordMovie
	MovieGrabScreen
	StopMovie
	DoScreenShot
common/server/audio.cs	ServerPlay2D
	ServerPlay3D
common/server/clientConnection.cs	GameConnection::onConnectRequest
	GameConnection::onConnect
	GameConnection::setPlayerName
	IsNameUnique
	GameConnection::onDrop
	GameConnection::startMission
	GameConnection::endMission
	GameConnection::syncClock
	GameConnection::incScore

continued

Team LRN

common/server/commands.cs	ServerCmdSAD
	ServerCmdSADSetPassword
	ServerCmdTeamMessageSent
	ServerCmdMessageSent
common/server/game.cs	OnServerCreated
	OnServerDestroyed
	OnMissionLoaded
	OnMissionEnded
	OnMissionReset
	GameConnection::onClientEnterGame
	GameConnection::onClientLeaveGame
	CreateGame
	DestroyGame
	StartGame
	EndGame
common/server/kickban.cs	Kick
	Ban
common/server/message.cs	MessageClient
	MessageTeam
	MessageTeamExcept
	MessageAll
	MessageAllExcept
	GameConnection::spamMessageTimeout
	GameConnection::spamReset
	SpamAlert
	ChatMessageClient
	ChatMessageTeam
	ChatMessageAll
common/server/missionDownload.cs	GameConnection::loadMission
	ServerCmdMissionStartPhase1Ack
	GameConnection::onDataBlocksDone
	ServerCmdMissionStartPhase2Ack
	GameConnection::clientWantsGhostAlwaysRetry
	GameConnection::onGhostAlwaysFailed
	GameConnection::onGhostAlwaysObjectsReceived
	ServerCmdMissionStartPhase3Ack
common/server/missionInfo.cs	ClearLoadInfo
	BuildLoadInfo
	DumpLoadInfo
	SendLoadInfoToClient
	LoadMission
	LoadMissionStage2
	EndMission
	ResetMission

continued

Team LRN

common/server/missionLoad.cs	LoadMission
	LoadMissionStage2
	EndMission
	ResetMission
common/server/server.cs	PortInit
	CreateServer
	DestroyServer
	ResetServerDefaults
	AddToServerGuidList
	RemoveFromServerGuidList
	OnServerInfoQuery
common/ui/ConsoleDlg.gui	ConsoleEntry::eval
	ToggleConsole
common/ui/GuiEditorGui.gui	GuiEditorStartCreate
	GuiEditorCreate
	GuiEditorSaveGui
	GuiEditorSaveGuiCallback
	GuiEdit
	GuiEditorOpen
	GuiEditorContentList::onSelect
	GuiEditorClassPopup::onSelect
	GuiEditorTreeView::onSelect
	GuiEditorInspectApply
	GuiEditor::onSelect
	GuiEditorDeleteSelected
	Inspect
	InspectApply
	InspectTreeView::onSelect
	Tree
	GuiInspector::toggleDynamicGroupScript
	GuiInspector::toggleGroupScript
	GuiInspector::setAllGroupStateScript
	GuiInspector::addDynamicField
	InspectAddFieldDlg::doAction
common/ui/LoadFileDlg.gui	FillFileList
	GetLoadFilename
common/ui/SaveFileDlg.gui	GetSaveFilename
	DoSACallback
	SA_directoryList::onSelect
	SA_filelist::onSelect

Team LRN

One last thing to remember about the common code: As chock-full of useful and important functionality as it is, you don't *need* to use it to create a game with Torque. You'd be nuts to throw it away, in my humble opinion. Nonetheless, you *could* create your own script code base from the bottom up. One thing I hope this chapter has shown you is that a huge pile of work has already been done for you. You just need to build on it.

Moving Right Along

In this chapter, we took a look at the capabilities available in the common code base so that you will gain familiarity with how Torque scripts generally work. For the most part, it is probably best to leave the common code alone. There may be times, however, when you will want to tweak or adjust something in the common code, or add your own set of features, and that's certainly reasonable. You will find that the features you want to reuse are best added to the common code.

As you saw, much of the critical server-side common code is related to issues that deal with loading mission files, datablocks, and other resources from the server to each client as it connects.

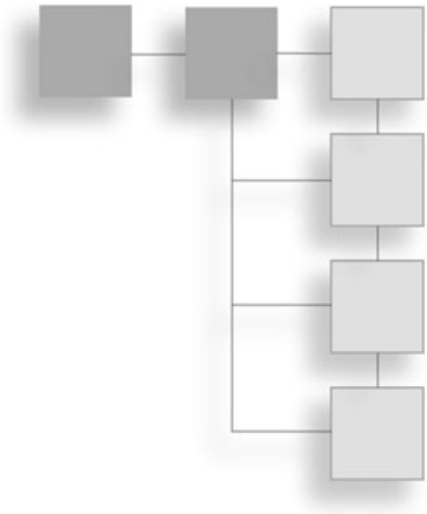
In a complementary fashion, the client-side common code accepts the resources being sent by the server, and uses them to prepare to display the new game environment to the user.

So, that's enough programming and code for a while. In the next few chapters, we get more artistic, dealing with visual things. In the next chapter, we will take a look at textures, how to make them and how to use them. We'll also learn a new tool we can use to create them.p

This page intentionally left blank

Team LRN

CHAPTER 8



Introduction to Textures

3D computer games are intensely visual. In this chapter we begin to explore the creative process behind the textures that give 3D objects their pizzazz.

Using Textures

Textures are probably the unsung heroes of 3D gaming. It is difficult to overstate the importance of textures. One of the most important uses of textures in a game is in creating and sustaining the ambience, or the look and feel of a game.

Textures also can be used to create apparent properties of objects, properties that the object shape doesn't have—it just looks like it does. For example, blocky shapes with jutting corners can appear to be smoothed by the careful application of an appropriate texture using a process called *texture mapping*.

Another way textures can be used is to create the illusion of substructure and detail. Figure 8.1 shows a castle with towers and walls that appear to be made of blocks of stone. The stone blocks are merely components of the textures applied to the tower and wall objects. There are no stone blocks actually modeled in that scene. The same goes for the appearance of the wooden boards in the steps and other structures. The texture gives not only the appearance of wood but the structure of individually nailed planks and boards. This is a powerful tool, using textures to define substructures and detail.

This ability to create the illusion of structure can be refined and used in other ways. Figure 8.2 shows a mountainside scene with bare granite rock and icefalls. Again, textures were created and applied with this appearance in mind. This technique greatly reduces the need to create 3D models for the myriad of tiny objects, nooks, and crannies you're going to

Team LRN

Figure 8.1 Structure definition through using textures.

Figure 8.2 Rock and icefalls appearance on a mountainside.

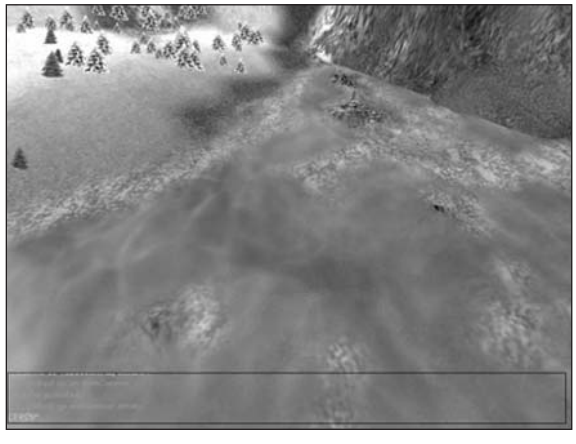


Figure 8.3 Shoreline foam and deepwater textures.

encounter on an isolated and barren mountain crag.

Textures appear in many guises in a game. In Figure 8.3 two different textures are used to define the water near the shoreline. A foamy texture is used for the areas that splash against rock and sand, and a more wavelike texture is used for the deep water. In this application the water block is a dynamic object that has moving waves. It ebbs and flows and splashes against the shore. The water textures are distorted and twisted in real time to match the motion of the waves.

Another area in a game where textures are used to enhance the ambience of a game is when they are used to define the appearance of the sky. Figure 8.4 shows cloud textures being used in a skybox. The *skybox* is basically the inside of a big six-sided box that surrounds your scene. By applying specially distorted and treated textures to the skybox, we can create the appearance of an all-enveloping 360-degree sky above the horizon.

We can use textures to enhance the appearance of other objects in a scene. For example, in Figure 8.5 we see a number of coniferous trees on a hillside. By designing the ground texture that occupies the terrain location of the trees appropriately, we can achieve the forest look we want without needing to completely cover every inch of ground with the tree objects. This is helpful because the fewer objects we need to use for such a purpose—

basically decoration—the more objects that will be available for us to use in other ways.

One of the most amazing uses of textures is when defining technological items. Take the Tommy gun in Figure 8.6, for instance. There are only about a dozen objects in that model, and most of them are cubes, with a couple of cylinders tossed in, as well as two or three irregular shapes. Yet by using an appropriately designed texture, we can convey much greater detail. The weapon is easily identifiable as a Thompson Submachine Gun, circa 1944.

Following the theme of technological detail, Figure 8.7 is another example. This model of a Bell 47 Helicopter (think *M*A*S*H*) shows two trick uses of textures in one model. The engine detail and the instrument panel dials were created using textures we've already seen. Now take a look at the tail boom and the cockpit canopy. The tail boom looks like it is made of several dozen intersecting and overlapping metal bars; after all, you can see right through it to the buildings and ground in the background. But it is actually a single elongated and pinched box or cube with a single texture applied! The texture utilizes the alpha channel to convey the transparency information to the Torque renderer. Cool, huh? Then there is the canopy. It is semitransparent or mildly translucent. You can obviously see right through it, as you should when looking through Perspex, but you can

Figure 8.4 Clouds in a skybox using textures.

Figure 8.5 Terrain accents.

Figure 8.6 Weapon detail using textures.

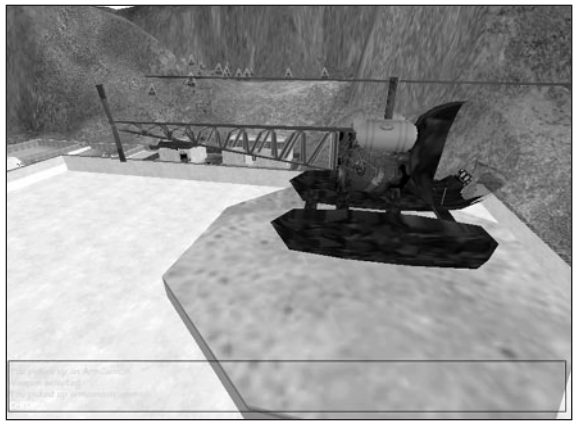


Figure 8.7 Vehicle detail and structure.

Figure 8.8 Player clothing, skin, and other details.

Figure 8.9 Distant objects.

still make out the sense of a solid glasslike surface.

Of course, technological features are not the only things that can be enhanced through textures. In Figure 8.8 the brawler about to enter the tavern is attired in the latest stylish leather brawling jacket. He is obviously somewhere around 40 years of age, judging by his classic male-pattern baldness, and the bat is a Tubettiville slugger. Okay, okay, the bat is a stretch, but if it were turned over 180 degrees, you would be able to see the Tubettiville logo, and then you could tell! Also note the use of the texture specifying the tavern name, named in honor of a famous *Delta Force 2* player, Insomniac.

Look at the moon in Figure 8.9. Look again, closer. Does it look familiar? It should, because the moon texture in that picture is an actual photograph of the full moon, taken outside my house with a digital camera and then used to generate the moon texture. The rest of the scene is generated using the Torque Engine with appropriate nighttime lighting parameters set.

I think by now you have a pretty good idea why I say that textures are the unsung heroes of 3D gaming. They really make a huge difference by conveying not only the obvious visual information, but also the subtle clues and feelings that can turn a good game into a great experience.

Paint Shop Pro

You are going to be creating your own textures as you travel through this book, and to do that you'll need a good tool for texture and image manipulation. Well, the good folks at JASC Software have graciously allowed us to include their *great* image processing tool, Paint Shop Pro, on the companion CD for you to use.

I've been using Paint Shop Pro for about 10 years, I think. In the early days, I only used it for converting file types, because that was about all it could do. Nowadays, though, it is a fully featured image processing and image generation tool, with scanner support, special effects and filters, image analysis statistics, and the whole nine yards.

First, you'll need to install Paint Shop Pro, if you haven't already run the Full Install from the CD.

Installing Paint Shop Pro

If you want to install only Paint Shop Pro, do the following:

1. Browse to your CD in the \PSP folder.
2. Locate the Setup.exe file and double-click it to run it.
3. Click the Next button for the Welcome screen.
4. Follow the various screens, and take the default options for each one, unless you know you have a specific reason to do otherwise.

Getting Started

To get this party rolling, we're going to just blast through and create a couple of textures that you can use later for whatever grabs your fancy. We'll cover just the tools and steps we need to get the job done. In a later section we'll cover the most common tools in more detail.

Creating a Texture

So, let's get down to brass tacks and create a texture from scratch. We'll create a wood texture using the built-in capabilities of Paint Shop Pro.

1. Launch Paint Shop Pro and select File, New.
2. A New Image dialog box opens up. Set the width and height dimensions to 128 pixels (see Figure 8.10) and click OK.
3. We now have a blank image to work with. Choose the menu item Effects, Texture Effects, Texture, and the Texture dialog box will appear, as in Figure 8.11.
4. In the visual list at the lower part of the Texture dialog box, select Woodgrain. You'll have to scroll down through the list to the bottom.

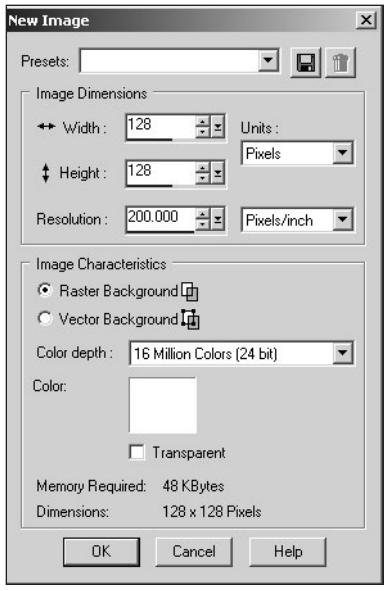


Figure 8.10 Creating a new blank image.

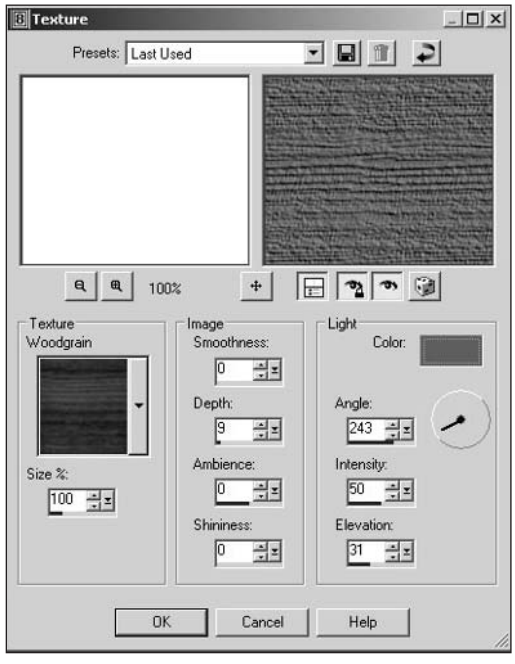


Figure 8.11 Texture dialog box with woodgrain texture.

5. Click on the color box at center-right. You will get a Color dialog box, as shown in Figure 8.12.

tip

Most of the image processing tools in Paint Shop Pro have an Auto Proof button that looks like this:

If you are using something newer than an old Pentium 100 computer, you should probably have this button pressed. It allows you to see the changes in your image as soon as you make them, rather than waiting until you click the OK button to close whichever dialog box you are using.

6. Change the value in the R (for red) box to 139, in the G (for green) box to 87, and in the B (for blue) box to 15. The H (hue), S (saturation), and L (light) boxes will automatically change to match.
7. Click OK to close the Color dialog box.
8. Change the other settings in the Texture dialog box to match those in Table 8.1.

Table 8.1 Texture Dialog Box Settings

Attribute	Value
Size	100
Smoothness	15
Depth	9
Ambience	0
Shininess	4
Angle	336
Intensity	50
Elevation	37

9. Click OK to close the Texture dialog box.
10. Now you should have a bona fide woodgrain texture, like the one shown in Figure 8.13, that you can use for things like walls, planks, ladders, the wooden stock on a weapon, barrels, and whatever else you can come up with.

11. With this texture, you can experiment with different image processing effects and touchup tools in Paint Shop Pro. Go ahead and give it a try.

Okay, that was so much fun, let's do another. This time we are going to tweak an image a bit searching for a specific look. The next texture will be a sort of rough-wall look that you might find on a painted cement block, or maybe a freshly poured sidewalk, or something like that. We'll call it the *sidewalk texture*, for convenience.

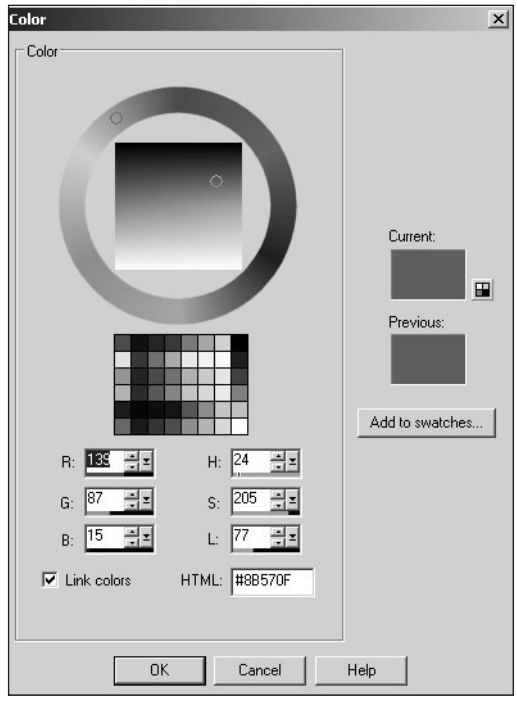


Figure 8.12 Color dialog box for woodgrain.

1. If it isn't still open, launch Paint Shop Pro.
2. Select File, New.
3. Set the width and height dimensions to 128 pixels and click OK. (Take another look at Figure 8.10 if you need to refresh your memory.)
4. Select the menu item Effects, Texture Effects, Texture, and the Texture dialog box will appear again, just like before, as depicted in Figure 8.11.
5. This time we'll do something a bit different from the previous image. Locate the Texture frame at center-left. Click on it to open a visual menu of textures, and choose Concrete. You should get a texture like the one shown in the boxes in the dialog box in Figure 8.14.

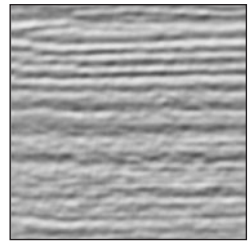


Figure 8.13 Woodgrain texture.

6. Click on the color box at center-right to get the Color dialog box again.
7. Using Figure 8.15 as a guide, change the value in the R box to 218, in the G box to 181, and in the B box to 110.
8. Click OK to close the Color dialog box.
9. Change the other settings in the Texture dialog box to match those in Table 8.1.
10. Click OK to close the Texture dialog box. You should get a new texture like the one shown in Figure 8.16.

    Now this texture is quite a bit darker than I want it to be. I'm looking for a gray with a hint of beige or tan color, so what we'll have to do is touch it up a bit. First, we want to brighten the highlights and, at the same time, darken the shadows a bit.

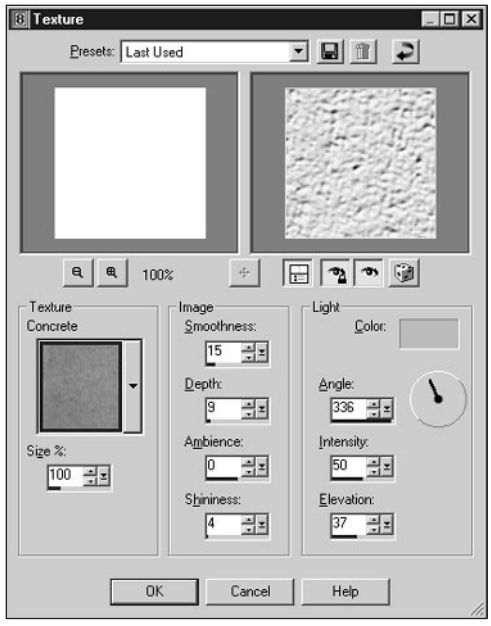


Figure 8.14 Texture dialog box with default preset.

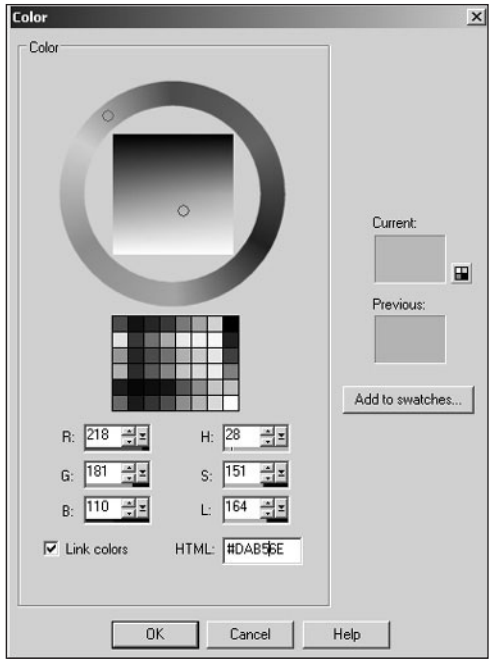


Figure 8.15 Color dialog box for sidewalk texture.

To do this, we'll use the Highlight/Midtone/Shadow tool.

11. Select Adjust, Brightness and Contrast, Highlight/Midtone/Shadow. You'll get the Highlight/Midtone/Shadow dialog box shown in Figure 8.17. Change your settings to match those in the figure.

    As you can see with Figure 8.18, the texture details now stand out in relief quite a bit more. This is goodness. However, the color is way too rich, or vibrant, for use as a sidewalk or wall texture, in my humble opinion.

    What we want to do now is tone down the richness of the color. We can do that by using the Hue/Saturation/Lightness tool. Now, we *could* have done this part using the Color tool when we created the color. And I tried! But it wasn't close enough, so using the Hue/Saturation/Lightness tool allows us to tweak the color in the direction we want.

12. Choose Adjust, Hue and Saturation, Hue/Saturation/Lightness to get the Hue/Saturation/Lightness dialog box, as shown in Figure 8.19.

13. Set the Hue to 0, the Saturation to −70, and the Lightness to 0, and then click OK.

    This will take the edge off the richness of the color quite a bit. If you look at Figure 8.20 and compare it with Figure 8.18, you can see the difference. You can use the Undo/Redo feature of Paint Shop Pro to compare your own versions of these images. Select the Edit, Undo and the Edit, Redo menu items to switch back and forth between the before and after versions of your own creation.

Now that the color is where we want it, let's roughen it up a bit. The texture is a bit too smooth, sort of like taffy. A sidewalk usually looks grainier. To do this, we'll add noise.

14. Choose Adjust, Add/Remove Noise, Add Noise. You'll get the Add Noise dialog box, as shown in Figure 8.21.
15. Set the Noise value to 19 percent.
16. Select the Gaussian button.
17. Check the Monochrome box. Compare Figure 8.22 with Figure 8.20, and you'll see the difference—the newly added roughness to the surface.

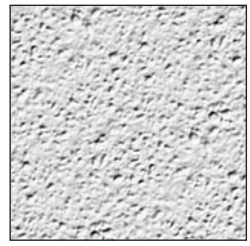


Figure 8.16 Initial sidewalk texture.

You should now have two images open in your Paint Shop Pro window: the first one being the woodgrain texture, and the other being the sidewalk texture. In the next section you'll learn how to save those images for later use.

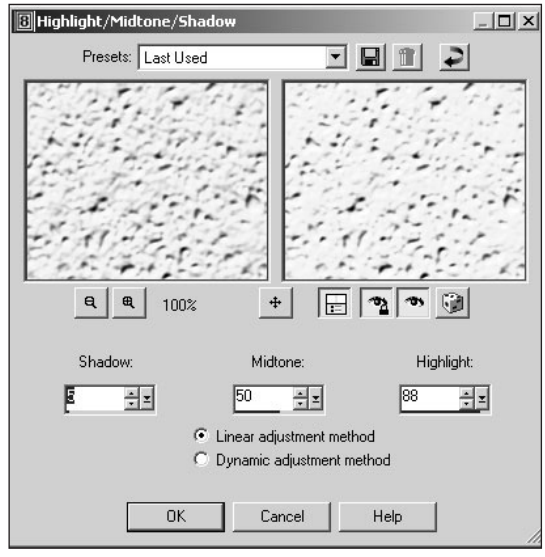


Figure 8.17 Highlight/Midtone/Shadow dialog box.

Working with Files

We want to get those images saved without any further ado, but first I want to show you something. You're going to launch the fps demo game that comes with the Torque Engine.

Launching the fps Demo Game

1. Leave Paint Shop Pro running and task switch (Alt+Tab) to the Windows desktop.
2. Using the Windows Explorer, browse into the C:\3DGPAi1 folder and then double-click on the fps demo shortcut.
3. When the GarageGames/Torque splash screen appears, click on the Start Mission button.
4. When the Mission dialog box appears, clear the Multiplayer check box.
5. Click on Scorched Planet to highlight that line.
6. Click on Launch Mission.

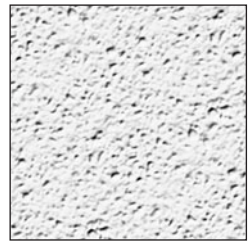


Figure 8.18 Enhanced highlight sidewalk texture.

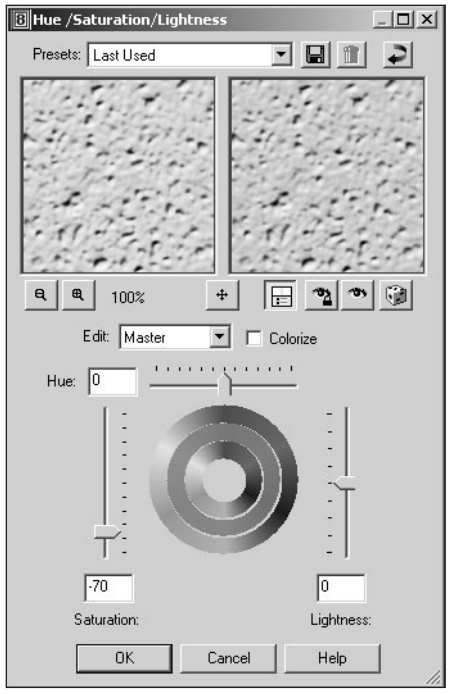


Figure 8.19 Hue/Saturation/Lightness dialog box.

When the game finishes loading you should get a view pretty well identical to the one in Figure 8.23, except it will be in color, of course. The floor will be bright orange, and the beams above you will be magenta. This odd coloring is deliberate—you are going to save your own textures in place of these two textures; this will make it easy for you to see if your changes have taken effect.

7. Resist the natural impulse to run around and blow things up. Instead, press the Escape key to exit the game (well, *try* to resist the natural impulse to run around and blow things up, anyway).
8. Click Quit.

Saving Texture Files

Okay, now that you have the "before" view recorded in your mind, we'll finally get to saving those images. Switch back to Paint Shop Pro now, and follow this procedure to save your files:

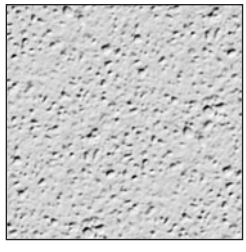


Figure 8.20 Desaturated sidewalk texture.

1. Click on the woodgrain image to bring it to the front (making it active).
2. Select File, Save As, and the Save As dialog box will appear.
3. In the Save As dialog box, make the type be JPG by scrolling through the Save As Type list and selecting JPG – JIFF Compliant.
4. Browse your way to C:\3DGPAi1\fps\data\interiors\evil1.
5. Name your file wood.jpg—the name must be exact.
6. Click OK.
7. You will get a dialog box that says "C:\3DGPAi1\fps\data\interiors\evil1\wood.jpg already exists. Do you want to replace it?" Click Yes.

Repeat steps 1 to 7 for the sidewalk image, using the name drk_cem.jpg.

Now, task switch back to the desktop and run the fps demo game again, just as you did before. When you spawn in the game you will now see the floor rendered with your new texture and the overhead beams rendered with the woodgrain texture you created. If either the floor or the beams look like they did in your "before" view, then you've

probably made an error in the file name or perhaps saved them in the wrong folder. Double-check your work, and everything should turn out fine.

Congratulations! Now you are an artist.

tip

The actual textures used for this platform object are saved in the evil1 folder as Original wood.jpg for the overhead beams and Original drk_cem.jpg for the floor. You can use the originals to replace your own textures if you want to see what they looked like.

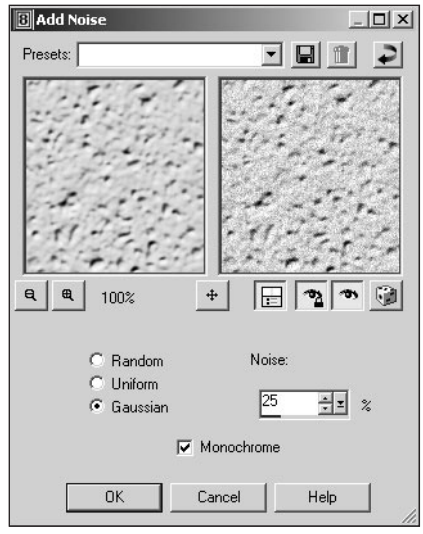


Figure 8.21 Add Noise dialog box.

PNG versus JPG

Paint Shop Pro supports many, many file types. If you select File, Save As, you'll get the Save As dialog box. If you click on the Save As Type combo box, you'll get a whopping great hockey sock full of available file types. There are two of particular interest to us: JPEG (*Joint Photographic Experts Group*) and PNG (*Portable Network Graphics*). In Windows, the JPEG format file extension is "JPG"; this has more common usage than "JPEG", so that's the term I will use.

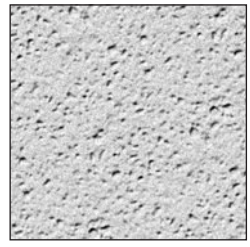


Figure 8.22 Final sidewalk texture.

When you save files in the JPG format, the images are compressed. The type of compression used is called a *lossy* compression. This means that the technique used to squeeze the image information into less space throws away some of the information. This is not necessarily a Bad Thing. The people who devised the JPG format were pretty clever and were able to specify rules that guide the software in what to keep, what to throw away, and how to modify the information. So although there is loss of information, the effect on the image is pretty negligible in most cases.

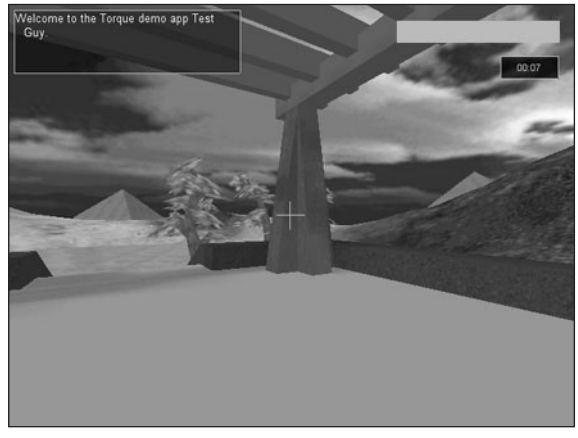


Figure 8.23 Spawn view in the fps demo game.

But there *is* an effect. Try this little experiment to see for yourself:

1. Create a new document the same way you did earlier with the two textures.
2. Make sure you have your foreground color set to black in the Materials palette (see Figure 8.24) and the background set to medium gray.

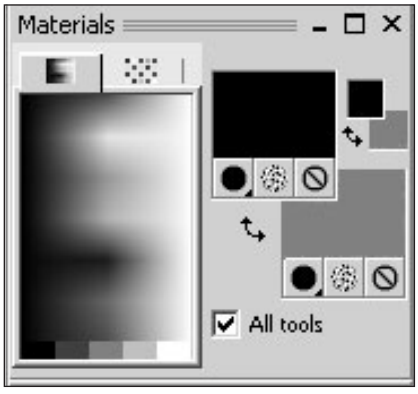


Figure 8.24 Materials palette.

tip

To change the foreground color, find the Materials palette at the right side, below the toolbars. There are two sets of overlapping color squares: a large and a small. Locate the upper-left large color square, which is called the Foreground and Stroke Properties box. Click this square and a Color dialog box will open. Select black and then close the dialog box by clicking OK.

The rectangle on the lower right is the Background and Fill Properties. You change it in the same way.

If you can't find the Materials palette, choose View, Palettes, Materials.

3. Choose Adjust, Add/Remove Noise, Add Noise to get to the Add Noise dialog box.
4. Select the Gaussian and Monochrome options.
5. On the right-hand side of the dialog box, set the Noise percentage to 60 percent.
6. Click OK, and the blank image will be filled with speckles and dots, similar to the image shown in Figure 8.25.

Figure 8.25 Noise image.

7. Next, select the Preset Shape Tool, the third last icon on the Tool palette.
8. Draw a shape in the middle of your image—any shape will do. Figure 8.26 will give you an idea.
9. Now select File, Save As, and the Save As dialog box will appear.
10. In the Save As dialog box, make the type be JPG by scrolling through the Save As Type list and selecting JPG – JIFF Compliant.
11. Give the file a name (I'll use "testing"), and save it wherever you can remember to find it again. Make sure you use a unique name so that you don't overwrite an existing file.
12. Click OK.
13. Now select File, Save As *again*, but this time select PNG format. This is an open standard file format used in many parts of the computer industry.
14. Close both of your image windows.

15. In the File menu, near the bottom is the Recent Files submenu, which contains a list of the names of recently used files. If you used the same file name that I did, you should see testing.png and testing.jpg in the list. Open both of them.
16. Use the Zoom tool, which is available via the top icon in your Tool palette (see Figure 8.27). Click on the icon, then select the Zoom tool from visual menu—the toolbar icon will change to reflect the chosen tool. Next, click once in each image to zoom it to a larger magnification. Right-click to zoom out again.
17. Compare the two images. You'll notice odd pixel artifacts around the lines of your shape in the JPG version that don't exist in the PNG version. Those artifacts are a result of the compression.

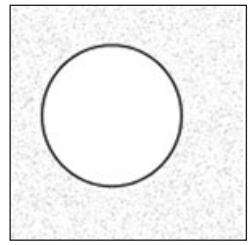


Figure 8.26 Noise image with shape.

See Figure 8.28 for a zoomed view of JPG image's artifacts in the upper portion of the figure, compared to the PNG image in the lower portion. The arrows in the JPG version point to some of the compression artifacts.

Figure 8.27 Tool palette, with tool names.

On top of all that, if you repeatedly open and save JPG files, the distortion will get worse each time you do it, as data is lost in the compression each time. You'll see it as a sort of smearing of colors around edges, especially in areas of high color contrast. It's similar to the messiness resulting from photocopying photocopies of photocopies.

So, if JPG has these artifacts, why use it? Because with more complex images, JPG files are usually smaller than PNG files. Go ahead and try for yourself. Maybe use the one in your texture example from earlier, like the sidewalk texture. When I save the final texture as JPG, I get a file size of 6,493 bytes. As PNG, I get 19,882 bytes!

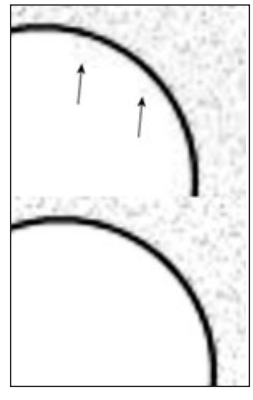


Figure 8.28 Noise image.

The smaller the texture files are, the more of them we can fit in a given amount of memory, and the more textures we can fit in memory, the richer the visual experience for our game.

Okay, so now you are wondering, why bother with the PNG file type, right? Well, there is a good reason for using PNG files, of

course. The PNG format supports a concept called alpha channels, and we will need to use *alpha channels* for some of our game images. Not all of them, but a few. So the rule of thumb will be to use JPG for all images except when we need to specify an alpha channel—then we use PNG.

Finally, here is an important workflow tip. Save all of your original image creations in the Paint Shop Pro native format: PSPIMAGE. When you create and save your images in PSPIMAGE format, it's a lot like having the original source code for a program. In PSPIMAGE format you can edit any text you've created in vector mode, modify objects and curves that are in vector format, move them around, and so on. You can then save them in the game format you need, PNG or JPG, when you need to.

Bitmap versus Vector Images

Paint Shop Pro uses image graphics in two different and complementary ways: bitmap graphics and vector graphics formats.

Bitmap images are also called *raster images*. Raster, the older term, is the pattern of lines traced by rectilinear scanning in display systems. Although it is not exactly the same as a bitmap image, it's the term that Paint Shop Pro uses to describe such images. In this book I will use the term *bitmap* for such images, except when quoting Paint Shop Pro tools or commands that use the word *raster*. Just remember that they essentially mean the same thing in this context.

A bitmap image is composed of pixels laid out on a grid. Each pixel represents a color value, one each for red, green, and blue. The weighting of each of these values determines the color of each pixel. In most image processing tools, if you increase the magnification of a bitmap image, you can see these pixels. They look like squares on the screen. A bitmap object is a collection of these pixels. An object is stored as a group of pixels with the color information about each pixel color. Pixels can be blended to create soft edges and smooth transitions between objects. Photographic images are usually rendered as bitmap images because the pixel format matches well to the way that photographs are made.

You should note that an image in bitmap format is resolution-dependent. You specify the resolution and pixel dimensions when you create the image. If you later decide to increase its size, you enlarge each pixel, which lowers the image quality.

A vector image is composed of procedural and mathematical instructions for drawing the image. As you encountered in Chapter 3, a vector is basically a line that has definite magnitude and direction. Vector objects in Paint Shop Pro are defined in a similar fashion. Each object in a vector image is stored as a separate item with information about its relative position in the image, its starting and ending points, and width, color, and curve information. This makes the vector format useful for things like logos, text fonts, and line drawings.

Team LRN

An image in vector format does not depend on the resolution. It can be resized without losing detail because it is stored as a set of instructions, not as a collection of pixels. Each time you display an image, you re-create it.

Now a computer monitor uses pixels to display an image, and most printers convert pixels to ink dots. Because of this, vector images are converted into pixel format when displayed on the screen or printed. However, when you close the image files and save them, they are saved in the vector format.

We will be doing most of our work with bitmap images. However, you may find that certain vector techniques can be used to create some image components, like curves and text. We'll create them as vectors and then convert them when required. Some of the tools we'll use allow you to create in either format. Most of the time we'll do it in the bitmap format when we are using fancy effects tools.

Creating Alpha Channels

Okay, so you are now able to perform the most important texture imaging operations, creating one and saving it. The next most important operation is the creation of alpha channel transparent sections of an image. Remember the helicopter tail boom?

There are other uses for alpha transparency, of course. Bitmapped GUI buttons are candidates: You may want a button that does not have straight sides and square corners. You can create irregular button shapes using transparent sections of your button image.

Another use for a bitmap with alpha transparency would be overlays on the GUI, such as health bars, status displays, and weapons crosshairs.

Let's take a look at an example of a bitmap with transparency.

Launch the fps demo game the same way you did when testing your textures earlier, but this time click the About button instead. You'll see the credits for the creation of the Torque Engine, with the nice round GarageGames logo. Let's change it!

1. In Paint Shop Pro select File, Open. Browse your way to C:\3DGPAi1\fps\client\ui and then find the file gglogo150.png and open it. We're going to work with this file. The original version has already been backed up as Original gglogo150.png for you, so don't worry about messing up the logo.

   Notice that there is no background color for this image; the areas outside the logo circle are filled with a gray-and-white checkerboard pattern. This pattern is the default Paint Shop Pro transparency pattern. The appearance of this pattern means that the alpha channel for the file has a value of 0 for each pixel of the image that coincides with the transparency pattern.

2. Now choose the Freehand Selection tool. (Refer back to Figure 8.27—you want the fifth tool icon from the top.) Click on the little black triangle to the right, and choose the Freehand Selection tool from the icon menu.
3. Use the Freehand Selection tool to select an irregular shape in the gglogo150 image, such as shown in part A of Figure 8.29.
4. Delete the selected portion, leaving a cutout in the logo, as shown in part B of Figure 8.29.
5. Now choose File, Save As to save your changes.
6. Launch the fps demo game and click on the About button.

You should now see that the nice round GG logo has a "hole" in it shaped like the selection you made, with the dialog box's gray background color showing through. Part C of Figure 8.29 shows what it looks like. If the dialog box's background color had been blue, that's the color you would see. Or if it had been some bitmap image, then parts of the image would show through the logo. The areas in the logo that were covered by the transparency pattern (as dictated by the alpha channel) are not drawn at all. Of course, your shape is probably not identical to the cutout shape I made.

This process, when carried out with most programs other than Paint Shop Pro, or even with some older versions of PSP, is a fairly complex activity. With PSP 8 it's pretty well a nonevent!

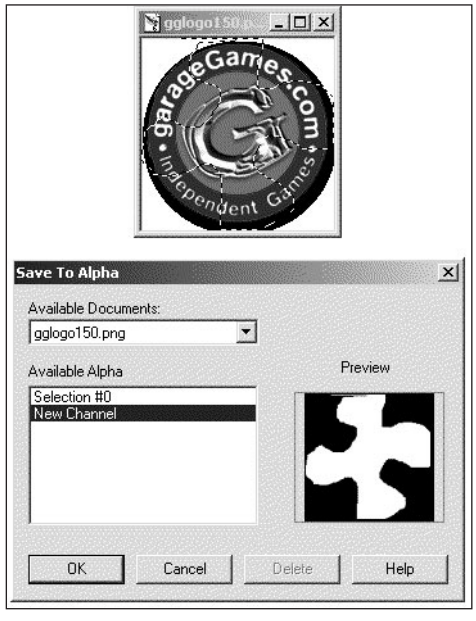


Figure 8.29 Creating an alpha channel selection.

Paint Shop Pro Features

I won't cover all of the features that Paint Shop Pro offers—and there are just a ton of them. What I'll do is cover those that I use the most when creating textures for games and present some of the most useful options and capabilities for those features.

Materials Palette

In Paint Shop Pro a *material* is a combination of color, gradient, pattern, and texture. The Materials palette provides a way to edit those attributes of either a foreground or background material.

The Materials palette is normally found docked on the right side, just below the toolbars.

On the Materials palette (see Figure 8.30), you will find the Colors tab and the Swatches tab.

The Colors tab is used to select either the foreground or the background color for the material. Simply move your cursor over the Colors tab body, and the cursor will change to the eyedropper icon. Set the foreground color by left-clicking on a color, and set the background by right-clicking on a color.

Swatches are custom materials you have created and saved for later use.

To the right of the tabs are the Foreground Properties and Background Properties boxes. When you choose a color from the Colors tab, the chosen color appears in the appropriate properties box. The upper-left properties box is for the foreground and stroke properties, and the lower-right one is for the background and fill.

Below each properties box are three buttons. From left to right these are the Style button, the Texture button, and the Transparent button.

- **Style button.** Indicates whether the color box is showing a color, gradient, or pattern.
- **Texture button.** Indicates whether a texture will be applied to the color.
- **Transparent button.** Indicates whether this color type (foreground or background) will be transparent.

To the right of the Foreground Properties and Background Properties boxes are the much smaller Foreground Color and Background Color boxes. Like the properties boxes, these reflect the current active colors. However, when you click in these boxes, you get a simple Color Selection dialog box and not the more complex Materials dialog box.

The All Tools check box allows you to indicate that the materials properties you have set apply globally to every drawing tool. When this option is not chosen, you will have to change the settings for each drawing tool you use.

If the Materials palette is not visible, choose View, Palettes, Materials to make it visible.

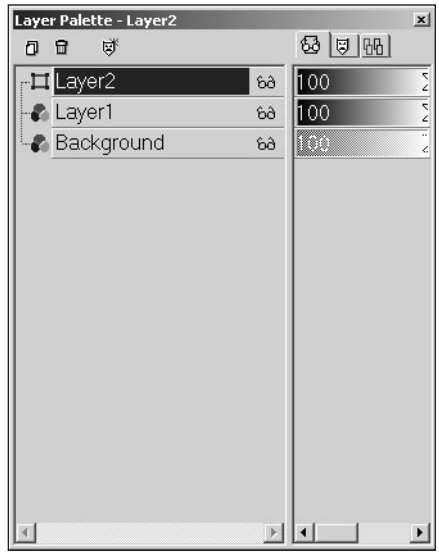


Figure 8.30 The Materials palette.

Layers

You can create four types of layers in Paint Shop Pro: raster layers, vector layers, mask layers, and adjustment layers. *Raster layers* are Paint Shop Pro's name for layers that contain pixel information (bitmaps). You can probably guess that *vector layers* contain instructions for drawing vector lines, shapes, and text. Vector layers can be added to images of any color depth, but not so with raster layers. *Mask layers* contain mask information, such

as the alpha channel stuff that we did earlier in this chapter. *Adjustment layers* contain color correction information. You use them to change the appearance of the underlying layers.

A newly created Paint Shop Pro image consists of one raster layer, always called the *background layer*. This is like the canvas of a painting; *every* image must have at least one layer. Paint Shop Pro permits up to 100 layers per image, providing your computer has sufficient memory.

A Layer palette, the Layer Properties dialog box, and various commands in the Layers menu are available for working with layers. For each layer, the Layer palette displays its order in the layer stack and current properties. For a vector layer, a button for each vector object you draw is also displayed. The Layer Properties dialog box shows the settings for any individual layer. The Layers menu contains the general options for creating, managing, and merging layers. It also lists the layers in the image.

tip

An image must be a grayscale image or a 24-bit color image to contain more than one bitmap layer or any adjustment layers. You might need to increase an image's color depth if you want to add these layers.

Creating Layers

To create a new layer, choose the Layers menu. You will see one menu item for each of the available layer types that can be created. If some aren't available, you need to increase the color palette for your image by choosing Colors, Increase Color depth. 16 million colors, if available to you, will give you access to all layer types.

The Layer Palette

The Layer palette provides quick access to many of the commands and options in the Layers menu and Layer Properties dialog box.

The Layer palette contains two panes (see Figure 8.31). Drag the vertical divider to change the size of each pane. The left pane of the Layer palette displays the names of layers and layer groups. The icon to the left of each name indicates the type of layer (background, raster, vector, mask, adjustment, group, or floating selection). When there are more layers than the palette can display, you can use the scroll bars on the right side to move the list of layers up or down.

The right pane of the Layer palette displays options for the layer, including visibility, opacity, blend mode, link set, and lock transparency. All settings on this pane apply to raster, mask, vector, and adjustment layers, but only the Visibility toggle applies to vector objects.

The Layer palette toolbar includes buttons for command tasks, such as adding a new raster or vector layer.

Team LRN

If the palette is not visible, choose View, Palettes, Layers to make the Layer palette visible.

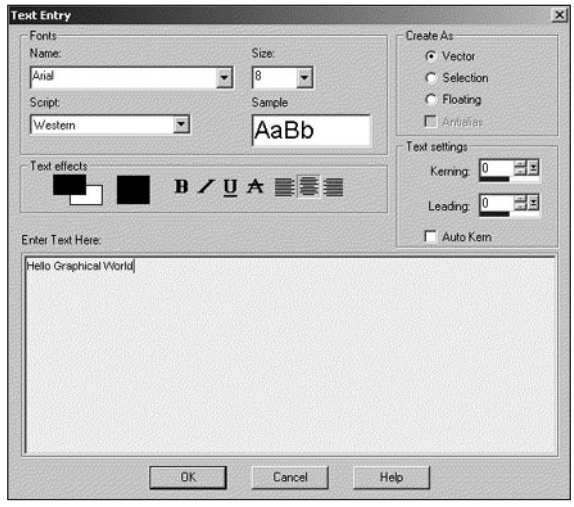


Figure 8.31 The Layer palette.

Layer Palette Buttons

Each layer in an image has a Layer button on the Layer palette where its name is displayed. To the left, an icon shows the layer type: raster, vector, mask, or adjustment. If you add an object to a vector layer, a plus sign ("+") appears next to its button and a Vector Object button appears underneath it. A Text Object button displays the text you typed. Line Object and Shape Object buttons display the types of lines and shapes, respectively. A Vector Object button is indicated by the "square and circle" icon to its left.

- Click a Layer button to make its layer active, or current. The button is highlighted to indicate that it is active. When you click a Vector Object button, its text appears in bold.
- A Layer button corresponding to a new layer appears on the palette and in the Layers menu list when you add a new layer. When you delete or merge a layer, its Layer button disappears.
- Right-click on a Layer button to get a pop-up menu of operations that can be performed on the layer. This menu is similar to the Layers menu.
- Right-click on a Vector Object button to get a pop-up menu of the editing operations available for the layer.
- Hover the cursor over a Layer button to see a thumbnail image of that layer, or over a Vector Object button to see a thumbnail of the object.
- Click the plus ("+") sign to the left of a Vector Layer button to expand the object list for a vector layer.

Layer Palette Controls

Several optional controls are available on the right side of the Layer palette. You will probably only need to use two of these controls: the Visibility toggle and the Opacity slider. The Visibility toggle merely indicates whether the selected layer or object should appear in the image. The Opacity slider indicates how much of the object or layer is visible. Opacity is the measure of how opaque an object is, so it is the opposite of transparency. A 100

percent opaque object or layer completely obscures any objects or layers that are behind it. A 50 percent opaque object would be like a ghost of an object, allowing a certain amount of the obscured objects or layers to show through.

Saving Layers

You can save images that contain raster, mask, and adjustment layers in either the PSPIMAGE format or Photoshop's PSD format. Both formats retain all the layer information for these two types of layers. However, images containing vector layers must be saved in the PSPIMAGE format to retain the vector information. When you save an image with vector layers in the PSD format, the vector layers are converted to bitmap layers.

Tool Options Palette

Every drawing tool has different adjustable settings that can be accessed via the Tool Options palette. The contents of this palette change according to which tool is being used.

Figure 8.32 shows the Tool Options palette in its default location, just below the toolbars. In the figure, the Paint Brush tool has been selected, so the Paint Brush options are being shown. The other parts of the window—the toolbars and menus—have been dimmed to make the Tool Options palette stand out.

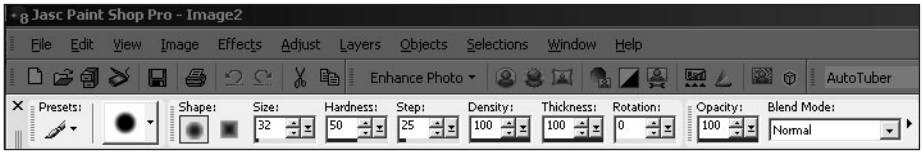


Figure 8.32 The Tool Options palette for the Paint Brush.

Paint Brush

The Paint Brush is one of the primary image creation tools. It is one of the brush tools that is associated with the seventh icon from the top of the Tool palette (refer back to Figure 8.27).

Using the Paint Brush

To "paint" in a freehand style, use the Paint Brush tool as follows:

1. Click the image where you want the brush stroke to start.
2. Drag the cursor while pressing the mouse button. (Press the right mouse button if you want to use the background color.)
3. Release the mouse button to end the brush stroke.

If you want to create straight lines, do this:

1. Click the image where you want the line to begin.
2. Press and hold the Shift key.
3. Click the image where you want to end the line (right-click to use the background color).
4. You can continue adding line segments of either color by clicking the mouse button.
5. Release the Shift key to end the line.

Paint Brush Tool Options

The Tool Options palette for the Paint Brush will appear in the upper toolbar when you select the Paint Brush. Figure 8.32 will help you locate the palette. The palette contains a number of optional attributes that you can alter; the most useful options are described in Table 8.2. These kinds of options are available for all tools on the Tool palette, although the specific options vary from tool to tool.

Table 8.2 Paint Brush Tool Options

Option	Description
Shape	Shape controls the shape of the brush tip: round, square, vertical, horizontal, right slash, and left slash. When the brush touches the image, it applies color in the selected shape. For example, if you click your image once with a square brush, the brush will apply a square-shaped area of color.
Size	Size is the width of the brush tip in pixels, from 1 to 255 pixels wide.
Opacity	Opacity controls how completely the color covers the image surface. Lowering the opacity is like diluting paint. At 100 percent opacity, the color covers everything; at 1 percent, the color is almost transparent.
Density	Density controls the amount of color the brush applies with each stroke. The density can be compared to the number of bristles in a real paintbrush—increasing or decreasing density adds or removes bristles. At 100 percent, the color covers the surface completely. As the density decreases, the amount of color applied with each stroke decreases. At 1 percent, only a few pixels of color appear.
Hardness	Hardness controls the sharpness of the brush edges. The harder the brush, the more defined the edges of color will be. At 100 percent, the stroke has sharply defined edges. As the hardness decreases, the brush edge softens.
Step	Step controls the spacing of the discrete pixels of color, or how frequently the brush tip touches the image during a stroke. It is a percentage of the diameter of the brush tip. At 100 percent, a size 30 brush tip touches the surface once every 30 pixels, and the brush shape is clearly defined; at 50 percent, the tip touches in the middle of the previous tip. As the step decreases, the brush tip touches the surface more frequently. Its outline becomes less noticeable, and the strokes appear smoother and denser.

Team LRN

Paint Shop Pro contains a selection of brushes with the preset tip shapes that approximate specific drawing tools: Paint Brush, Pen, Pencil, Marker, Crayon, Chalk, and Charcoal. If you use one of these brushes, all attributes except the size and step values are set automatically. You will find these presets at the left end of the Tool Options palette, under the label "Presets".

The numeric edit controls used in the palette provide a variety of ways to change the settings. You can type a number in the box, click the spinner controls, drag the meter bar, or press the pop-out button and drag the slider.

Table 8.3 describes options that are available only for the Paint Brush tool and not for the other brush tools.

Table 8.3 Paint Brush Special Brush Options

Option	Description
Continuous Stroke	Normally, the Paint Brush tool simulates the effect of repeated brush strokes. Each time you move the brush over the same area of the image (or layer), you apply more paint. When the Continuous Stroke check box is selected, the brush applies paint once, and repainting an area has no effect.
Wet Look	With this option selected, the Paint Brush tool simulates wet paint, with soft color inside and a darker ring near the edge. To see the effect, lower the Hardness setting to some value less than 100 percent.

Air Brush

The Air Brush is used to paint the same way you would with an airbrush or a spray can. It is one of the three brush tools that are associated with the seventh icon from the top of the Tool palette (refer back to Figure 8.27).

tip

To confine the brush painting to a specific area, use the Selection tool or the Freehand Selection tool to make a selection before painting. Then the brushwork will only be applied within the selected area. This is a handy technique to avoid "overspray" with the Air Brush.

Freestyle Airbrush Painting

Freestyle airbrush painting is a technique where there are no restrictions on brush movement. To paint something in a freestyle manner:

1. Click the image where you want the brush stroke to start.
2. Drag the cursor while pressing one of the mouse buttons (press the right mouse button to use the background color).
3. Release the mouse button to end the brush stroke.

Team LRN

Straight Airbrush Lines

We can restrict the movement of our airbrush in order to make consistent lines, by doing the following:

1. Click the image where you want the line to begin.
2. Press and hold the Shift key.
3. Click the image where you want to end the line (press the right mouse button to use the background color).
4. Keep adding line segments of either color by clicking with the left or right mouse button.
5. Release the Shift key to end the line.

Clone Brush

The Clone Brush is the eighth tool from the top of the Tool palette. Use the Clone Brush to copy part of an image to another location. You can clone within an image, between bitmap layers, or between two grayscale or 24-bit color images. For example, if a photograph has a flaw against a multitoned or multicolored background, such as skin or cloth, you can use the Clone Brush to copy the background over the flaw.

When you use the Clone Brush, you work with two image areas: the *source* area, which is the area you copy from; and the *destination* (or *target*) area, which is the area you copy to. The destination can be within the same image or in another image of equal color depth.

Clone Brush Options

The Clone Brush shares many options with the other brushes. However, one option is unique to the Clone Brush and has a big effect on how it operates: the Aligned Mode option.

With this option enabled, the source area moves with the brush each time you release the mouse button. When you release and then reclick the mouse button, the brush resumes cloning the image relative to the distance from the source area.

With this option disabled, the source area does not move when you release the mouse button. Each time you release and then reclick the mouse button, the starting point for cloning returns to the source area.

There is also the Sample Merged option. With this option enabled, the brush will clone all visible data rather than just the data from the current layer. If not enabled, only the data on the current layer when the source point was defined is cloned.

tip

When you clone from one image to another, make sure that the two images have the same color depth before you begin.

Cloning

To use the Clone Brush, follow this procedure:

1. Position the cursor over the part of the image that you want to copy. Set the source area by right-clicking the source area once. Your computer beeps to indicate that you have selected the source area.
2. To place the cloned image on a specific layer or in a selection, select that layer or area now. Paint Shop Pro only clones within the selection.
3. Move the cursor to the area to which you want to start copying the image. This can be within the same image or in another image of the same color depth.
4. Press and hold the left mouse button to make the crosshairs appear over the source area to indicate which pixel you are copying.
5. Drag the mouse to clone from the source area to the destination area.
6. Release the mouse button to end the brush stroke.
7. To resume cloning, start over at step 5. Remember that the location of the source area depends on the clone mode.

Eraser

Use the Eraser to replace colors in an image with the background color or with a transparency. When you drag the Eraser across a bitmap layer, all the pixels in its path become transparent. When used on a background, the Eraser produces a different effect. It acts like a paintbrush, replacing the existing color with the current foreground or background color.

The Eraser retains the information it has removed from a layer. To restore the erased image, you can right-click and drag the Eraser over the transparent areas.

To use the Eraser:

1. Activate the Eraser by clicking its button in the Tool palette. (The Eraser is in the seventh slot from the bottom. Refer back to Figure 8.27.)
2. Use the Tool Options palette to configure the Eraser tip for your needs.
3. Drag the cursor across a layer to erase the color.

Selecting

There are several ways to select things in Paint Shop Pro. When working with bitmap images on raster layers, you select pixels with one of two tools, Selection or Freehand. When working with vector layers, you select objects with the Vector Object Selection tool.

Selection Tool

Use the Selection tool to create a selection in a specific preset shape. The Selection tool is located in the fifth slot from the top of the Tool palette (refer back to Figure 8.27). As with other tools, you have a range of tool options to use (see Table 8.4).

Table 8.4 Selection Tool Options

Option	Description
Selection Type	Choose one of the selection shapes from this drop-down box. Your choices are rectangle, square, rectangle or square with rounded corners, ellipse, circle, triangle, pentagon, hexagon, octagon, one of two star shapes, and one of three arrow shapes.
Mode	Normally you would use the Selection tool in Replace mode, where each time you use the tool, you create a new and different selection. You can use Add mode if you want each selection you make to be added to the previous selection. Remove mode removes the area of each selection from a previous selection. You will find, however, that it is probably best to just use Replace mode and press the Shift key to temporarily invoke Add mode or the Control key to temporarily invoke Remove mode.
Feathering	Feathering controls the sharpness of a selection's edges. By fading a set width (in pixels) along the edges, it produces a smooth transition between a selection and the surrounding area. The feathering value is the width of the transition area in pixels. A higher feathering value creates softer edges by feathering more pixels. Feathering is useful when pasting a selection. The fading helps the selection blend into the background.
Anti-alias	Anti-aliasing is similar to feathering, but more precise. It produces a smooth-edged selection by partially filling in pixels along the edge, making them semitransparent. If anti-aliasing is not applied, the edges of a selection can appear jagged. Anti-aliasing is useful when combining images and when working with text.

To make a selection:

1. Click the Selection button on the Tool palette.
2. Place the cursor on the image.

tip

To create a rectangular, square, or rounded rectangular or square selection, place the cursor at a corner of the area you want to select.

To create a circular or elliptical selection, place the cursor at the center of the area you want to select.

To create a selection using the other shapes, place the cursor at a point that would form the corner of an imaginary rectangle enclosing the shape.

3. Click and drag the mouse until the selection is the size you want. As the cursor moves, a line appears to indicate the border of the selection.
4. Release the mouse button. The selection border becomes a marquee.

Freehand Selection Tool

The Freehand Selection tool shares the same slot in the Tool palette as the Selection tool. It makes selections with three types of borders:

- Irregularly shaped borders
- Point-to-point straight borders
- Borders between areas of contrasting colors or amount of light

You can change the Freehand Selection tool's selection shape from its Tool Options dialog box by choosing one of the three selection shapes from this drop-down box. This works exactly the same as with the Selection tool.

The Freehand Selection tool has the same options as the Selection tool, with one addition: Smoothing. This option smoothes sharp corners and jagged lines. The higher the value, the more smoothing is done.

Use the Freehand Selection tool to draw the outline of the selection as follows:

1. Click the Freehand Selection icon on the Tool palette.
2. On the Tool Options palette, set the needed options.
3. Move the cursor over the image.
4. Click the image at a point that you want to become the border of the selection.
5. Drag the cursor to create an outline of the area you want to select. Be careful here—don't release the mouse button while creating your selection, or you may end up selecting stuff you don't want.
6. If you release the mouse button, start again, add to the selection using the Shift key, or remove part of the selection using the Ctrl key.
7. When the line encloses the selection, release the mouse button. The line becomes a marquee indicating the border of the selection.

Masks

A *mask* is a grayscale image that you apply to a layer. You can use it to hide and display parts of the layer, to fade between layers, and to create special effects with precision. Masks can be created from selections, alpha channels, and images.

A mask can cover a layer completely or with varying levels of opacity. The gray value of the mask determines how much it covers. Where it is black, it completely masks the layer; where it is white, it leaves the layer unmasked; where it is gray, it produces a translucent effect.

Team LRN

All masks are created and edited in a grayscale bitmap mode. Therefore, all tools and image processing features that work on grayscale images work on masks. The tools that can be used in either vector or bitmap mode (Drawing, Preset Shapes, and Text) work only in their bitmap modes when editing masks.

A mask works the same way with a vector layer as it does with a bitmap layer. It can be linked to a layer, which moves it with the layer. If a mask is not linked to the layer, moving the layer's content will not affect the position of the mask.

Because a mask is grayscale, you can save it with the image in an alpha channel or as a separate image on a hard disk. The texture for the helicopter canopy in Figure 8.7 was created as a grayscale mask saved in an alpha channel! Also, you can load a selection as a mask and a mask as a selection from an alpha channel.

Remember that you must choose the mask layer you are editing by selecting it on the Layer palette before painting so that you edit the mask, not the image. When you are editing the mask, the colors available to you become those of a grayscale image. When you click a foreground or background color box, the grayscale palette appears. When you switch to a nonmask layer, the active color boxes return to their previous colors.

tip

Any painting tool or effect that can be applied to a grayscale image can be applied to a mask.

When you edit a mask, you change either the areas or the level of masking. For example, painting over an object to mask it changes the area, while making a gradient fill edits the degree of masking. A gradient fill is where we fill a shape with colors that gradually fade from one color to another. Usually we use a grayscale gradient when making masks. For example, we might use a gradient that transitions from dark gray to white. The dark gray masks more than the lighter grays. As the gradient approaches white, there is less masking effect.

Creating a New Mask Layer

To create a new Mask Layer:

1. On the Layer palette, click the layer for which you want to create a mask.
2. Choose Layers, New Mask Layer and then choose the type of mask:
   - **Show All.** This type shows all underlying pixels. All pixels of the mask are white. Paint the mask with grays or black to hide portions of the underlying layers.
   - **Hide All.** This type hides all underlying pixels. All pixels of the mask are black. Paint the mask with white or grays to show portions of the underlying layers.

The mask layer and the selected layer are added to a new layer group. The mask layer applies to the selected layer only.

3. Use the painting tools to alter the masked area.
4. To view the mask on the image, click the Mask Overlay toggle on the Layer palette, at the far right of the layer properties palette for the mask layer you created.

tip

To apply a mask to all underlying layers, drag it from the layer group to the main level of the Layer palette.

Creating a Mask from a Selection

To create a mask from a selection:

1. Use the Selection, Freehand Selection, or Magic Wand tool to make a selection on a raster or vector layer in the image.
2. Do one of the following:
   - **Mask the selected area.** To do so, choose Layers, New Mask Layer, Hide Selection.
   - **Mask the unselected area.** To do so, choose Layers, New Mask Layer, Show Selection.

   The mask layer and the selected layer are added to a new layer group. The mask layer applies to the selected layer only.
3. Use the painting tools to alter the masked area if needed.
4. To view the mask on the image, click the Mask Overlay toggle on the Layer palette.

Creating a Mask from an Image

To create a mask from an image:

1. Open the image that you want to use for the mask.
2. Select the image in which you want to create the mask layer.
3. On the Layer palette, click the layer you want to mask.
4. Choose Layers, New Mask Layer, From Image to open the Add Mask From Image dialog box.
5. In the Source window drop-down list, select the image to use for the mask.
6. In the Create Mask From Group box, select one of the following:
   - **Source luminance.** The luminance value of the pixel color determines the degree of masking. Lighter colors produce less masking, darker colors produce more masking, and transparent areas completely mask the layer.

- **Any nonzero value.** Transparent areas completely mask the layer. There is no gradation to the masking. Pixels with data (opacity of 1 to 255) become white pixels in the mask, and transparent pixels become black in the mask.
- **Source opacity.** The opacity of the image determines the degree of masking. Fully opaque pixels produce no masking, partially transparent pixels create more masking, and transparent pixels produce full masking.

7. To reverse the transparency of the mask data, select the Invert Mask Data check box. Black pixels become white, white pixels become black, and grays are assigned their mirror value.
8. Click OK.

   The mask layer and the selected layer are added to a new layer group. The mask layer applies to the selected layer only.
9. To view the mask on the image, click the Mask Overlay toggle on the Layer palette.

Scaling Images

You may need to scale your image, making it larger or smaller. To do this, use the Resize feature, as follows:

1. Choose Image, Resize.
2. Select a method for resizing the image. The Resize dialog box presents you with two sizing method options:
   - **Pixel Dimensions.** Select a new size by choosing a new measurement in pixels or one based on a percentage increase or decrease from the original.
   - **Actual/Print Size.** Select a new size by changing the resolution or the page dimensions. Note that the two are linked.
3. Enter the new measurements in the Width and Height boxes of the Pixel Dimension panel. In the Actual/Print Size panel, you can also change the resolution.
4. In the Resample Using box, select the type of resizing for Paint Shop Pro to apply. There are five choices:
   - **Smart Size.** Paint Shop Pro chooses the best algorithm based on the current image characteristics.
   - **Bicubic Resample.** Paint Shop Pro uses a process called *interpolation* to minimize the raggedness normally associated with expanding an image. As applied here, interpolation smoothes out rough spots by estimating how the "missing" pixels should appear and then filling them with the appropriate color. It produces better results than the Pixel Resize method with photorealistic images and with images that are irregular or complex. Use Bicubic Resample when enlarging an image.

Team LRN

- **Bilinear Resample**. Paint Shop Pro reduces the size of an image by applying a method similar to the Bicubic Resample. Use it when reducing photorealistic images and images that are irregular or complex.
- **Pixel Resize.** Paint Shop Pro duplicates or removes pixels as necessary to achieve the selected width and height of an image. It produces better results than the resampling methods when used with hard-edged images.
- **Weighted Average.** Paint Shop Pro uses a weighted-average color value of neighboring pixels to determine how newly created pixels will appear. Use this type when reducing photorealistic, irregular, or complex images.

5. In an image with more than one layer, select the Resize All Layers check box to resize the entire image. Leave the box unchecked to resize only the active layer.
6. To change the proportions of the image, select the Maintain Aspect Ratio of check box and type a new ratio for the image width. *Aspect ratio* is the relationship of the image's width to height. By default, the Aspect Ratio box displays the image's current aspect ratio.
7. Click OK to close the dialog box and apply the changes.

tip

After resizing, many images can be improved by using the Sharpen or Soften filters.

Bilinear and Bicubic resampling are available only for grayscale images and 24-bit images. To resample an image with a lower color depth, do the following:

1. Increase the image's color depth.
2. Resize the image.
3. Reduce the image's color depth to the original depth.

Rotating

Use this feature to rotate a selection, a layer, or an image of any color depth.

To rotate an image, a layer, or a selection:

1. Choose Image, Rotate, Free Rotate.
2. Select the direction of rotation by clicking the Direction's option button or its text. Right is clockwise, and left is counterclockwise.
3. Set the degrees of rotation in quarter-circle increments (90-, 180-, or 270-degree option) or rotate by any other amount by typing the value in the Free box.
4. To rotate every layer in a multilayer image, select the All Layers check box. Clear the check box to rotate only the current layer. When this check box is selected or

when the image consists of a single background layer, the canvas size changes to accommodate the rotated image.

5. Click OK to close the dialog box and rotate the image.

Image Sizes

Use the Change Canvas Size dialog box to change the dimensions of the image. Because the current background color is used for pixels added to the background layer of an image, select a background color before opening the dialog box.

Changing the canvas size changes the size of the background, without changing the size of the image or any layers that may be in the image.

To change the image resolution, use the Resize dialog box, not the Canvas Size dialog box.

To change the size of the image canvas:

1. Choose Image, Canvas Size.
2. In the Dimensions panel, enter a new size (in pixels) for the image in the New Width and New Height boxes. You can type a number or use the spin controls. The current width and height are displayed for your reference.
3. Use the Arrow buttons in the dialog box to indicate how you want the image to be placed in the newly dimensioned canvas.
4. Use the edit boxes to enter precision placement information that will supersede the Arrow buttons, if needed.
5. After positioning the image, click OK to close the dialog box and apply the changes.

Text

There will be times when we want our game textures and images to contain text. Now we could use the paintbrush and try to write out our text in a freehand fashion. However, there just so happens to be a very handy Text tool available with lots of capabilities.

Looking back to Figure 8.27, you'll find that the Text tool is the fourth one from the bottom of the Tool palette.

Go ahead and create a new blank image, then select the Text tool, and click in the center of your image. You'll get the Text Entry dialog box.

Using Figure 8.33 as your guide, you'll note that the first and most obvious feature is the Text Edit box. You can type many lines of text in here; there is a limit, but it is high. I have been able to enter 32 lines of 128 characters each with no penalty other than a little slowdown in response time.

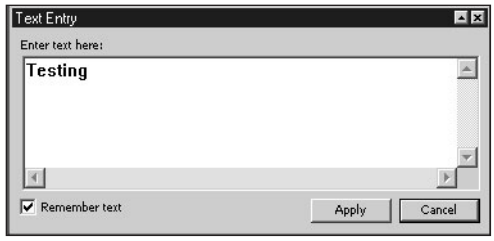


Figure 8.33 The Text Entry dialog box.

Let's create some text so that we can look more closely at the available options.

1. In the Font section of the Tool Options palette, you can scroll through the Name, Stroke Width, and Size lists and click to make your selection.
2. The Create As section is where you choose the creation mode for the text. You can select one of three modes:
   - **Vector.** A vector object on a vector layer.
   - **Selection.** An empty selection on the current layer.
   - **Floating.** A selection floating above the current layer.

   You should usually create your text as vector type. This allows you to easily edit and manipulate the text at any time. You can collapse the vector text into your bitmap image when you are happy with using the Layers, Merge, Merge All (Flatten) menu item.
3. By selecting the Anti-alias check box, you can soften the jagged edges that can appear on bitmap text; this feature partially fills in pixels in the jaggy spots. You can only do this with grayscale and 24-bit color images, however. (You should create your textures as 24-bit color images unless you have a specific reason to do otherwise, anyway.)
4. To select a color for the text, use the Materials palette.
5. Add emphasis effects to the text by using the Font Style options. When you choose an effect, it is applied to the next character you type. Change the effects of specific characters by highlighting them and then clicking the effects buttons. You can select from four style effects: Bold, Italic, Underline, and Strikeout.
6. The Alignment buttons are to the left of the Font Style buttons. These buttons set how the ends of multiple lines of text line up with each other: left, center, or right paragraph alignment. These settings affect the entire text in a paragraph and can't be changed for individual lines. Different paragraphs can have different alignments.
7. In the Text Settings section at the far right end of the Tool Options palette, set specific *leading* (space between lines) and *kerning* (space between letters) by clearing the Auto Kern check box and typing values in the Leading and Kerning edit boxes.

   If you cannot see the Text Settings section, you will see a small black triangle at the right end. Click on this triangle to make the Tool Options palette shift over, revealing the tools that couldn't fit in your window on the right-hand side.

8. In the text box, type the text you want to add to the image. Click the background (away from the text) to remove the highlighting and view the text. Note that the text does not display kerning and leading changes you have made.
9. Click OK to close the dialog box; this places the text in the image at the location where you had clicked with the Text tool.

There you go. You should now have your text sitting in your image, centered on the spot where you clicked the Text tool, highlighted with a dashed-line box with the sizing handles on the corners.

Moving Right Along

In this chapter we had our first peek at the world of textures. As the book unfolds, we will examine the uses for textures in more detail.

Then we took a detailed look at a powerful imaging tool that we can use to create and edit textures—Paint Shop Pro. As you have seen, Paint Shop Pro has a very complete feature set.

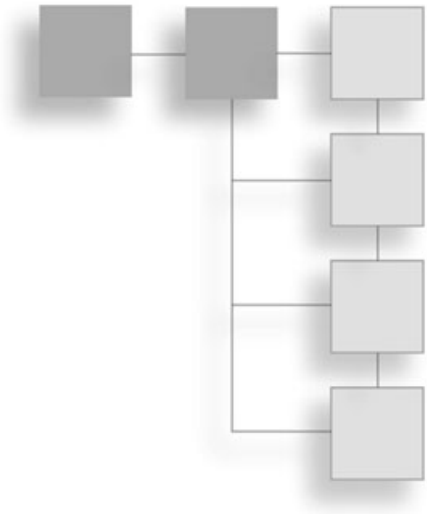
In the following chapter, we will expand our understanding of using textures in game development by learning how to *skin* objects, such as player models and vehicles.

Team LRN

This page intentionally left blank

Team LRN

CHAPTER 9



Skins

Skins are special textures used in games. The quality that separates skins from regular textures is that they typically wrap around the shape of a 3D model. It is fairly obvious that 3D monsters and player-characters would have texture skins, but the term can also apply to automobiles, wheelbarrows, mailboxes, rowboats, weapons, and other objects that appear in a 3D game.

Typically, skins are created after a model has been unwrapped, so that the skin artist knows how to lay the skins out in the UV template. We're going to do the process a bit backward, simply because we should stay on topic with Paint Shop Pro and textures until we've covered the topic sufficiently.

In our case here, it isn't a big issue anyway, because I'm providing you with UV templates from previously UV unwrapped models to work with.

UV Unwrapping

UV unwrapping is a necessary function prior to skinning a model. Consider it part of the *modeling* process in the context of this book. However, in this chapter we'll deal with the *texture* processing part of skinning a model and use models I've provided on the CD. Later you'll create and skin your own models and do the unwrapping and other things. We'll cover how the unwrapping works in more detail then.

When we want to apply textures to 3D objects, we need a system that specifies where each part of a texture will appear on which parts of a model. The system is called *U-V Coordinate Mapping*. The U-coordinate and the V-coordinate are analogous to the X- and Y-coordinates of a 2D coordinate system, though they're obviously not exactly the same.

Team LRN

Imagine (or you can actually try this at home yourself) taking a closed cardboard box and slicing it open along the edges. Then lay the whole thing out flat on the kitchen table, with no parts overlapping parts. There, you've unwrapped your box. Now get out your crayons and draw some nifty pictures on it. Then glue it all back together again to make a box. I think you get the idea.

With UV unwrapping we apply the technique to some complex and irregular shapes, like monsters and ice cream cones.

The Skin Creation Process

When we begin the skinning process, we will have a bare, unadorned 3D model of some kind. For this little demonstration, we'll use a simple soup can (see Figure 9.1). It's a 12-sided cylinder with a closed top and bottom (end caps). Each side face is made up of two triangles, and the end caps are made of 12 triangles each, for a total of 48 triangles. Nothing too special here.

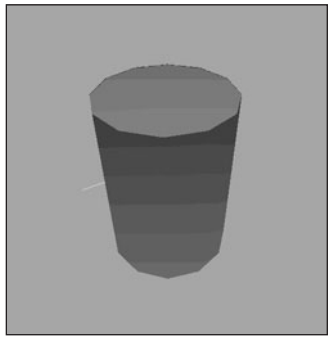


Figure 9.1 The victim—a simple can of soup.

Using the UV Unwrapping tool, we have to basically spread all our faces out over a nominally flat surface (see Figure 9.2).

We save the image of the UV template, plus we save the original model file, because the UV Unwrapping tool will have modified the UV coordinates for the objects in the model, and we can save those changes to the file so that the modeling tool can read them back in again.

Then we import the unwrapped image with the lines indicating the face edges into an image processing tool like Paint Shop Pro and apply whatever textures, colors, or symbols we need, such as shown in Figure 9.3.

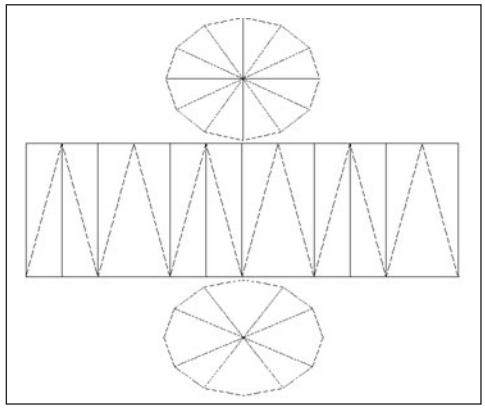


Figure 9.2 Laying it all out—the unwrapped can.

Notice that for textures I simply created markings and re-created a simple label. For the top of the can I made some circular text, and for both end caps I made a circular pattern that represents the ridges you often find in those places on tin cans. The image file has now officially become a skin for the can!

The final step is to import the new skin into the modeling program (or the game) and view the results, as in Figure 9.4.

The part of the process we will focus on in this chapter is the activity shown in Figure 9.3, the actual creation of the textures on the UV

template, so that it can be later used as a skin for models.

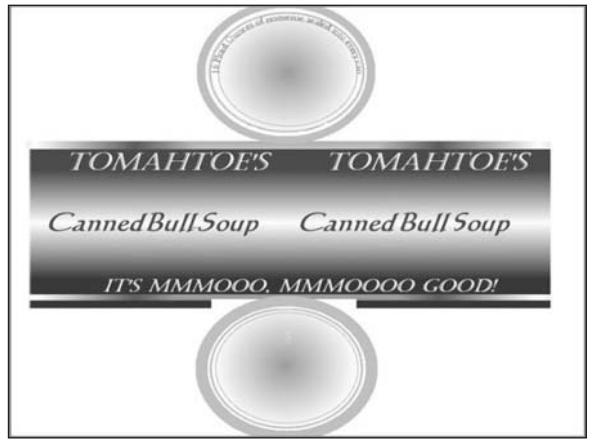


Figure 9.3 After applying textures.

Making a Soup Can Skin

So let's dive right in and create a skin. We can use the bare model of the soup can I showed you in the last section. The procedure has quite a few steps—more than 30—so we'd better roll up our sleeves and get to it.

The Soup Can Skinning Procedure

This is how you skin a soup can:

1. Open C:\3DGPAi1\resources\ch9\can.bmp in Paint Shop Pro. This file contains the UV mapping template.

Figure 9.4 Aha! Not such a simple can anymore. Nutritious, too!

tip

Remember when I said that the only file types we would need to use are JPG and PNG in the last chapter? Well, that was sort of a lie, though not quite—you see, the only file types we will be using for *making games* will be those two types. However, the UVMapper program outputs its UV mapping templates as one of two types: BMP (Windows bitmap) or TGA (Targa) format. So I've picked BMP to be our standard UV mapping template format. We won't be creating any game files in this format, however.

2. Choose Image, Increase Color Depth, 16 Million Colors. You need to do this to get access to the full palette.
3. Save the file as C:\3DGPAi1\resources\ch9\mycan.psp. This way you can re-use the layers over and over at later times if necessary. Make sure you save your work often as you follow the steps, in case you royally mess up, like I frequently do.
4. Right-click the Layer palette (see Figure 9.5), and then choose New Raster Layer.
5. Accept the default settings and click OK.
6. Click Raster 1 to make that layer active.
7. Click the Preset Shape tool, third icon from the bottom of the Tools toolbar on the left.

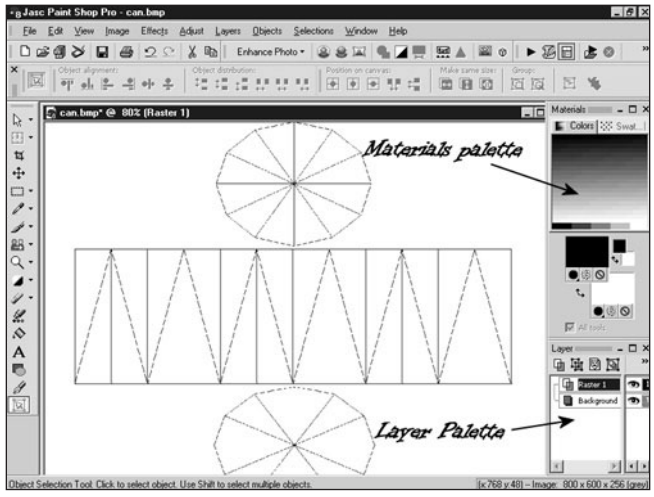

Jasc Paint Shop Pro - can.bmp
File Edit View Image Effects Adjust Layers Objects Selections Window Help
can.bmp* @ 80% (Raster 1)
Materials palette
Layer Palette

Figure 9.5 Materials and Layer palettes.

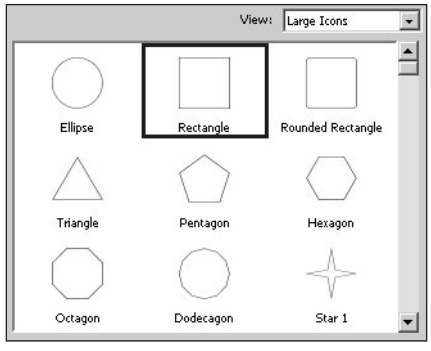


Figure 9.6 The Shapes List.

8. In the Tool Options palette (if it is not visible, choose View, Palettes, Tool Options to make it visible), click the Shapes List icon. The Shapes List will appear, as in Figure 9.6.
9. Choose the Rectangle from the Shapes List control; the list will go away automatically.
10. If the Create As Vector box on the Tool Options palette is checked, clear it.
11. If the Retain Style box on the Tool Options palette is checked, clear it.
12. In the Materials palette, locate the Foreground and Stroke Properties control. It is the upper-left box in the larger pair of boxes in the Materials palette.
13. Along the bottom of the Foreground and Stroke Properties are three buttons. Click the one on the far right with the No Entry icon, called Transparent. Now that it is depressed, the foreground or stroke of the object you draw will be transparent.
14. Click the Background and Fill Properties control, the lower-right control of the larger pair of boxes in the Materials palette. When the Color dialog box opens, select a bright red, and make sure it appears in the box marked Current. Then click OK to close the dialog box.
15. Now draw a rectangle that completely covers the rectangle containing the triangles in the middle of your mycan.psp image, as in Figure 9.7.
16. Now use the Background and Fill Properties control to set the fill to white, and draw a thin white rectangle across the middle of the red rectangle you just made (see Figure 9.8).

    So now you have your basic red-and-white pattern on the sides of the can. If you look at Figure 9.3 again, you'll notice that the red blends into the white gradually. There are several ways to do this. For example, you could have used a gradient fill

in the rectangles you created. But you're going to use another method, one that is more of a touchup technique.

17. First, you're going to select the sides of the can. Use the Square Selection tool (fifth icon down on the Tools toolbar) to select the entire rectangle that encloses all of the red and white rectangles you've just made, but only those areas. Use Figure 9.9 as a guide.
18. Next, soften the transition between the red and the white. Choose Adjust, Softness, Soft Focus. You will get the Soft Focus dialog box, as shown in Figure 9.10.
19. Set all of the values in the boxes to 100 percent, except for Halo size. Set Halo size to 70 percent, and then click OK to close the box. You'll see the edges between the red and the white go blurry.
20. Repeat the last two steps—choose Adjust, Softness, Soft Focus, and then make sure all values are set to 100 percent (except Halo Size, which is set to 70 percent) and close the dialog box. There now—you have your blended pattern.
21. Next you'll want to add metal lips to the top and bottom of the can sides. Do this by creating a thin light gray rectangle all the way across the top and another at the bottom, as shown in Figure 9.11. The black arrows indicate the location of the lip line.
22. Now you'll want to create the surface texture for the ends, or lids, of the can. Once again, select the Preset Shape tool.
23. In the Tool Options palette, click on the Shapes List icon.
24. Choose the Ellipse from the Shapes List control.
25. This time make sure that the Create as vector box is checked.

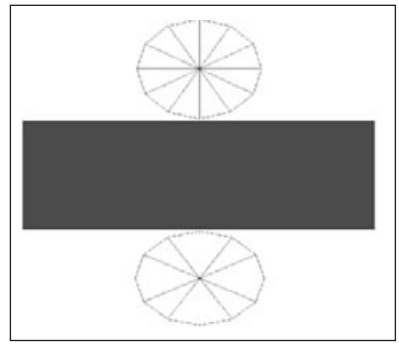


Figure 9.7 The red rectangle.

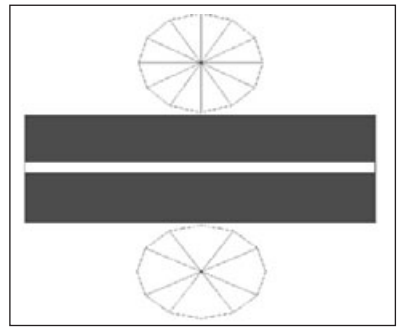


Figure 9.8 The white rectangle.

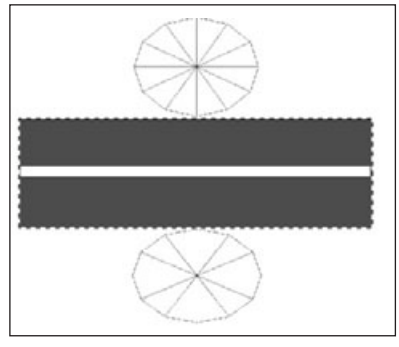


Figure 9.9 Selecting the mapped sides of the can.

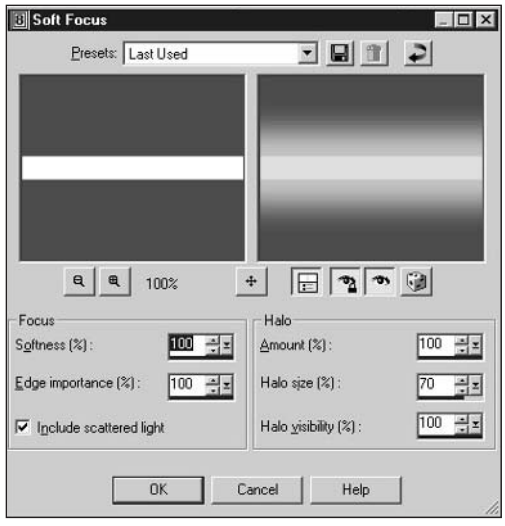


Figure 9.10 Soft Focus dialog box.

26. Set the foreground color to a light gray and the background color to transparent, using the same technique you used when drawing the rectangles earlier.
27. Draw a series of concentric ellipses in both the top and bottom list shapes (see Figure 9.12). Make sure to leave a sizable gap between the inner ellipse and the one next to it.

    Also use a line width of about 15 for the outermost ellipse, 2 for the middle ellipse, and 4 for the innermost one. The ellipses are drawn from the center out. You may need to fiddle a bit before you get it right.

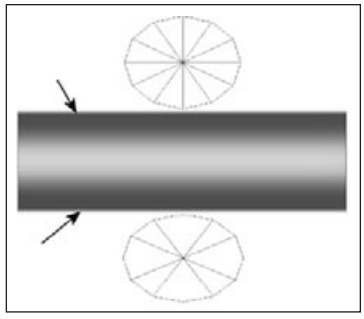


Figure 9.11 Adding the metal lips.

tip

You can adjust the size of your vector object ellipses by grabbing the little black handles on the shapes while the object is selected. If it isn't selected, use the Object Selection tool (the bottom tool of the Tools toolbar) to select your ellipse by clicking on it.

You can also drag your ellipses around this way. Press Ctrl+Left Arrow to nudge the object a bit to the left. Press the Ctrl key with the other arrow keys to nudge the object in other directions.

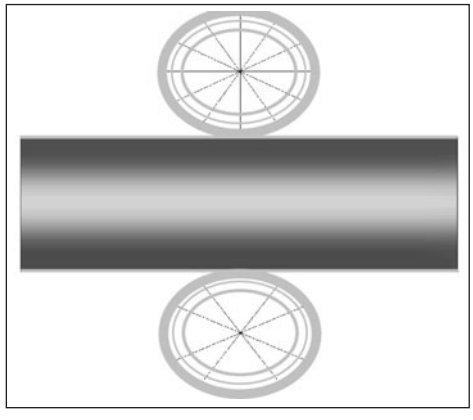


Figure 9.12 Adding the ridges.

28. Next, select the Text tool, which is fourth from the bottom of the Tools toolbar. If you have any objects already selected, deselect them by clicking the Object Selection tool on an empty part of the image before selecting the Text tool.
29. As you move the Text tool cursor around, it will have an icon like the one on the left in Figure 9.13. When you move it over a vector object, the cursor will change to the one on the right in Figure 9.13. Move the cursor over the top part of the innermost ellipse object in the upper set of concentric rings.

30. Now click on that object; you will get the Text Entry dialog box. Select the font you want from the Tool Options palette.
31. Use a font size of 12 and make sure you set the stroke width to 1.0.

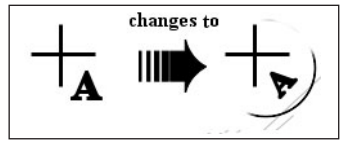


Figure 9.13 Text tool cursors.

32. Type in your text, something like **16 Fluid Ounces**.
33. Make sure you have vector set in the Create As control; then click Apply.
34. Voilá! You will have text that follows the curve of the ellipse around in an arc.
35. Now add your main label text using the Text tool. You can type whatever you want and position it wherever you want.
36. When you are finished, save your file one final time as C:\3DGPAi1\resources\ch9\mycan.psp. This is your source file.
37. Next, save your work as C:\3DGPAi1\resources\ch9\mycan.jpg. Make sure you've selected the "JPEG – JIFF Compliant" type in the Save As dialog box when you do this.
38. You will get an alert saying that it will have to save the file as a merged image and asking if you want to continue. This is expected because the JPG format doesn't support layers. Click Yes.

Testing the Soup Can Skin

Congratulations! You've made your first skin! I suppose now you want to see what it looks like all wrapped around a tin can and everything. Okay, so do this:

1. Browse your way to C:\3DGPAi1 and then double-click on the Show Book Models shortcut.
2. The Torque Engine will fire up with a special program called the *Show Tool*. Click on Load Model.
3. Find mycan.dts and load it.
4. Presto! That's your skin on that there soup can! Good job!
5. You can admire your creation in all its splendor by using the navigation keys to move the can back and forth and rotate it about the various axes. See Table 9.1 for the Show Tool key commands.
6. You can view my original soup can skin by loading the soupcan.dts model.

Table 9.1 Torque Show Tool Navigation Commands

Key	Description
A	rotate left
D	rotate right
W	bring closer
S	move farther away
E	rotate top backward
C	rotate top forward

Making a Vehicle Skin

Okay, soup cans are cool and soup hits the spot, too. But now that lunch break is over, let's move on to something a bit more serious. Many people are going to have vehicles in their games, and the Torque Engine does quite a nice job of supporting vehicles. We'll be making our own vehicles later, but because this chapter is on creating skins, let's make a skin for some kind of vehicle.

For a bit of a tease, let's take a look at a vehicle that is already included in the Torque demo using the Show Tool.

1. Browse to C:\3DGPAi1 and click on the Show Racing Models shortcut. This is *not* the same shortcut as Show Book Models.
2. Click Load Shape.
3. From the list, select buggy.dts, which is near the bottom.
4. Zoom in using the navigation keys and take a gander at the buggy chassis. Pretty cool, huh? Notice that it has no wheels. In Torque we model the wheels separately, so that we can model the suspension action of the vehicle more accurately.

The Dune Buggy Diversion

Okay, okay. I knew you would want to do this, so I'll show you how to test-drive the dune buggy *in-game*, as long as you promise to come back here after you've tired out your driving fingers. People tend not to learn quite as well when they are pouting.

1. Browse to C:\3DGPAi1 and click on the Run racing Demo shortcut.
2. Click on Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Car Race Track from the mission list.
5. Click Launch.

Team LRN

6. After the game loads, have at it! You probably should switch to Chase view by pressing the Tab key—there's more to see. See Table 9.2 for the keyboard controls.

Table 9.2 Torque Racing Demo Controls

Key	Description
mouse	steering left or right
W	accelerate
S	brake
Tab	toggle from first- to third-person viewpoint
Escape	exit the game

The Runabout Skinning Procedure

Okay, now that the old adrenaline is pumping, let's get back to making skins. We're going to create a skin for a less ambitious, but still pretty cool, vehicle—the runabout. It's a fictional creation of mine that's a convergence of memories of summers spent reading Doc Savage pulp stories and memories of a classic 1936 Auburn Boattail Speedster that I saw at a car show once as a teenager.

1. Open C:\3DGPAi1\resources\ch9\runabout.bmp in Paint Shop Pro. This file contains the UV mapping template.

   This time, I've unwrapped the object differently. If you recall, the soup can was completely unwrapped so that each individual face was lying flat. This time I unwrapped the runabout by showing only the separate objects (except the cab) from one particular view—the side or the top.

   By doing this, I can treat each of these objects as symmetrical, with the hidden side being simply a mirror image of the visible side. This is another valid technique, but it does have some pitfalls, which we will encounter. The advantage of using this approach is that it saves on image editing time, because only half of the objects' surfaces need to be given textures.
2. Select the Pen tool (second from the bottom). Set the line color to red and the fill to transparent.
3. Select Segment Type to be Point to Point on the Tool Options palette, as shown in Figure 9.14.

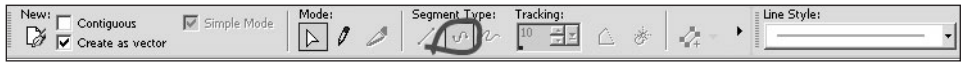


Figure 9.14 Point to Point Segment Type button.

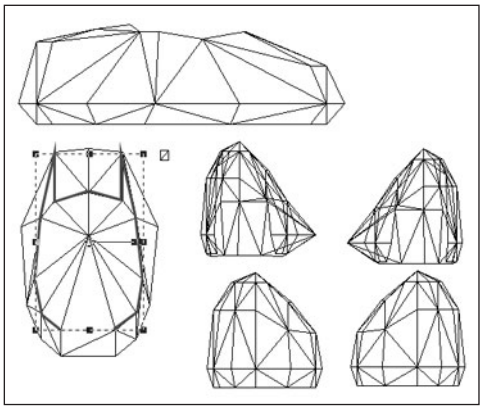


Figure 9.15 Tracing the cab roof.

4. Using the selected object with the thicker lines in Figure 9.15 as a guide, trace a shape around the cab triangles that cover the roof and the C-pillar. You click at a start location and then click again at each place around the cab where the object's line will change direction. Each click defines a *node* of the object.
5. When you have made the last node of your object, click on the Object Selection tool (bottom icon of the Tools toolbar). The object will now be surrounded by a rectangle with reshaping handles (black squares).
6. If you are not happy with the shape you've made, select it again with the Object Selection tool (if it isn't already selected), and then click the Pen tool. In the Tool Options palette in the Mode section, click on the little white arrowhead icon, which is the Edit Mode button (see Figure 9.16).

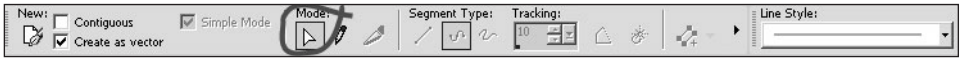


Figure 9.16 Edit Mode button.

7. Move your cursor down to your object and reshape the object by grabbing and moving the little square node handles (see Figure 9.17) that are located at the places where you clicked when creating the object—these are the nodes.

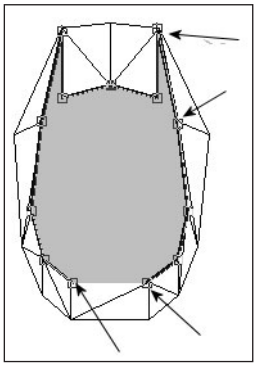


Figure 9.17 Arrows indicate the editable node handles.

8. Once you've finished, change the fill color of the object you've just created to the color of your choice, and make the line color transparent. By drawing the object in this way, you don't have the fill color obscuring your template while you trace the outline. Using red line color when you are drawing helps to differentiate between the template's black lines and the lines you are making.
9. After you've finished with the cab roof, right-click anywhere on the drawing to deselect the object you just drew, and then set the line color back to red and the fill color back to transparent.
10. Next, draw an outline around the entire cab template, following the outside edges. When finished, click the Object Selection tool.

11. Set the fill of this new object to a dark green-gray color, and set the line color to transparent. The new object should now obscure both the earlier roof object plus the rest of the cab template.
12. Select the new object with the Object Selection tool, and then choose Objects, Arrange, Send to Bottom. This will move the last object you created to be positioned underneath the first object you made that covered the roof. See Figure 9.18 to see what the before and after should look like.
13. Next, create a new raster layer by right-clicking the Layer palette and choosing New Raster Layer. Accept the defaults and click OK in the dialog box that appears.
14. Now find the Paint Brush tool (the seventh one down), but instead of just clicking the icon, click the little black triangle or arrow on the right side of the icon. A pop-up icon list will appear, from which you should select the Air Brush.
15. Set your foreground color to be the same blue (or whatever) color you used for the cab roof.
16. Now set your Air Brush tool options to match those in Table 9.3.
17. Use the Air Brush to spray over the outline of the car's body, as shown in Figure 9.19. Remember to make sure that the new raster layer you created (probably called Raster 1) is selected in the Layer palette, otherwise your airbrushing will be applied to the wrong layer.

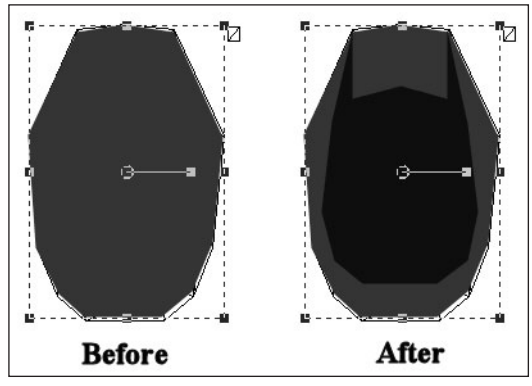


Figure 9.18 Before using Send to Bottom, and after using it.

Table 9.3 Air Brush Settings

Setting	Value
Shape	Round
Size	32
Hardness	9
Step	1
Density	100
Thickness	100
Rotation	0
Opacity	35
Blend Mode	Normal
Rate	5

Team LRN

Figure 9.19 Spray-painting the body base color.

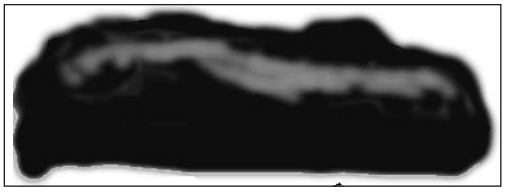


Figure 9.20 Spray-painting the accent color.

18. Now change to a light blue foreground color, and set the Air Brush size to 16 and the opacity to 12.
19. Spray on the accent color, as shown in Figure 9.20. When I did it, I sprayed one long line from left to right, and then I went back and used short spurts to make the line more irregular.
20. Next you'll apply a fancy racing stripe. Select the Pen tool, and change it to Drawing Mode by clicking on the little pencil icon next to the Edit Mode arrow we used earlier.
21. Make sure that Create as vector is set in the Tool Options palette, and then select Freehand for the Segment Type, also in the Tool Options palette.
22. Set the line color to be yellow.
23. Draw a squiggly line, with a shape like the one in Figure 9.21, from left to right on the car body. You can edit the shape of the line by twiddling the node handles, just like you did earlier when we made the cab roof. To make your line look more like Figure 9.21, choose the Object Selection tool, and deselect the line.

So, there you have it—the car's body paint job is done. Notice that we used a different approach than we did with the cab. It just goes to show that there's more than one way to skin a *cat*…er, *car*! I meant car! Honest.

Well, I guess it's time to get back to work. The last bits left are the four wheel-well, fender thingies. We'll do these in a fashion similar to the way we did the cab.

24. Using Figure 9.22 as a reference and using the Pen tool set to Point to Point Segment Type, draw an outline of the upper part of the upper-left fender thingy and fill it with the basic blue we've been using.

Figure 9.21 Adding the racing stripe.

25. After creating the last node, choose the Object Selection tool and make sure the object you just made is selected.
26. Choose Edit, Copy to copy the object to the Clipboard.
27. Choose Edit, Paste, Paste As New Layer. The new object will be pasted in its own layer.

28. Choose Image, Mirror. This will cause your new object to face left instead of right, or vice versa. This is why I used Paste as New Layer, instead of Paste as New Vector Selection. When you use the Image, Mirror menu or the Image, Flip menu, all the objects on the current layer are affected. By creating a new layer with just the one object, you avoid this problem.
29. Place your new copy of the object over the upper-right fender thingy template, adjusting it with the Pen tool in edit mode if necessary.

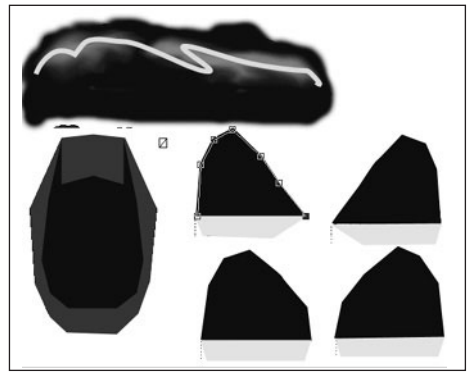


Figure 9.22 The fender thingies.

30. Repeat steps 24 to 29 for the bottom two fenders.
31. Now repeat all of steps 24 to 30, but this time create the fender skirts with a fill color of yellow, which stylishly matches the racing stripe.
32. When you are finished, save your file one final time as C:\3DGPAi1\resources\ch9\myauto.psp. This is your source file.
33. Now save your skin as C:\3DGPAi1\resources\ch9\myauto.jpg.
34. Once again, you will get an alert saying that the software will have to save the file as a merged image, and asking if you want to continue. Choose Yes.

Testing the Runabout Skin

Now it's time to take our little creation out for a spin around the block, so to speak. We'll use the Show Tool just like we did with the soup can.

1. Browse your way to C:\3DGPAi1 and then double-click on the Show Book Models shortcut.
2. The Torque Engine will fire up the Show Tool. Click on Load Model.
3. Find myauto.dts and load it. A fine job, indeed! Notice the lack of wheels, the same as with the dune buggy you looked at earlier.
4. Don't forget to use the navigation keys to move the car back and forth, and rotate it about the various axes. See Table 9.1 for the Show Tool key commands.
5. You can view my original runabout skin by loading the runabout.dts model.

Unfortunately, we'll have to wait until the later modeling chapters before we can take the runabout out for a real test drive. That's okay, though—we've plenty to do in the meantime!

Making a Player Skin

Now for the Big One—the player skin, or more accurately, the *character* skin, because the following section could apply equally as well to computer-controlled characters sometimes called AI (*Artificial Intelligence*) players or NPCs (*Nonplayer Characters*).

The character we'll use as the basis for this section is one affectionately called the *Standard Male Character*. He was created to be the base model for derivatives to be used in the *Tubettiworld* game that is currently in development at Tubetti Enterprises.

Figure 9.23 shows an early prototype of the Standard Male Character striking a heroic pose in the wilderness, confronting his, um, well, some trial or tribulation, I guess. This character began life as a concept sketch I did while nestled in front of a roaring fire on vacation in the Laurentians. My wife told me what the character should look like, and I sketched him about a hundred times until she was happy with it (see Figure 9.24).

I sent the concept artwork to a talented young man who goes by the name Psionic (http://www.psionic3d.co.uk) on the Internet, and he created the original model prototypes for me. The model came out pretty well, but as I said, the character in Figure 9.23 was an early prototype. The main issue was the skin color—it was too pasty. But that was soon fixed. We have since used that model to generate variations in gender, build, ethnicity, and animation sets, mostly by modifying skins, but with some model changes as well—especially for the female versions.

The point here is that for all of your serious artwork, models, skins, and so on, it's a good idea to create concepts beforehand—on paper or digitally, it doesn't matter. This way you have a tool to communicate the idea that you have in your head. It may take weeks or months to get a model completed, and it can happen that you stray unacceptably far from your original concept. It's especially important to have concept artwork if you want to sell a game idea to build a team and recruit talent to help you. If they can go away with a few pictures in their minds of what your dream is, it will help you a great deal.

Figure 9.23 The *Tubettiworld* Standard Male Character model rendered by the Torque Engine.

The Head and Neck

Now on with the show. Take a look at Figure 9.25. This is the unwrapped UV template for

the Standard Male. I've labeled the various parts in the picture to help identify what goes where. The file C:\3DGPAi1\resources\ch9\player.bmp has the proper template in it (though without the labels) for you to work with. Let's get started.

1. As before, open the template file, this time called C:\3DGPAi1\resources\ch9\player.bmp, save it somewhere as a PSP file, and work with that.
2. Create a new raster layer and name it "Skin" in the New Raster Layer dialog box. You are going to create a lot of raster layers in this procedure—make sure you label each one as I indicate.
3. Using whichever technique you like best, cover the entire face and neck part of the template with a flesh color, as in Figure 9.26. (I use the RGB values shown in Table 9.4 for a basic flesh color. Of course, you are free to twiddle the numbers to get something you like.) Make sure you apply your color to the skin layer and not to the background layer that holds your template.

Figure 9.24 Concept artwork for the Standard Male Character.

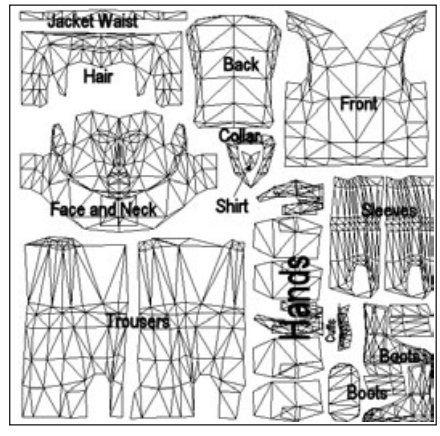


Figure 9.25 UV template for the Standard Male.

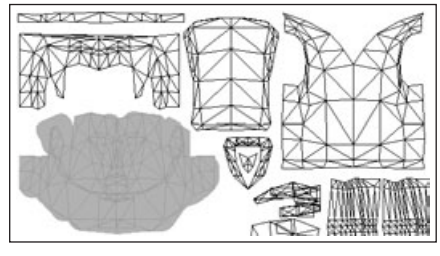


Figure 9.26 Basic flesh tone applied to the skin layer.

tip

Normally you can draw a vector object directly to a vector layer or create it as a raster on a raster layer. The problem is that once you've drawn it, you can't adjust it. The solution is to draw your object as a vector on a new vector layer and then add it to a raster layer when you have finished with it. You do this by setting all the layers you aren't interested in to invisible (click on the eye icon next to each layer in the Layer palette). You want to have only two layers visible: the vector layer of the object you just made and the raster layer you want to add it to. Then you choose Layers, Merge, Merge Visible. Your vector layer will be converted to raster form and added on top of the raster layer. Then just set all the rest of your layers to be visible again by clicking on their eye icons one more time.

Table 9.4 Flesh-Tone RGB Settings

	Color Component	Value
Basic	Red	215
	Green	165
	Blue	95
Shadow	Red	183
	Green	133
	Blue	83
Highlight	Red	247
	Green	187
	Blue	107

tip

In Figure 9.26 you can see the lines of the UV template through the skin layer's flesh color. Do this by reducing the opacity of the skin layer to about 95 percent or so. In the Layer palette, slide the Opacity slider left until it gets to the value you want. The lower you set the opacity, the more you can see of the layer beneath it—however, the less your skin layer's colors will look like their actual settings.

4. Now comes a bit of magic. You need to get some basic skin shading done next. There is a highlight and shadow image template that I like to use to get the basic head shades in place. Figure 9.27 shows the template, and you have a copy of it (C:\3DGPAi1\resources\ch9\hilite.png) that you can use for your own purposes. Open this file in Paint Shop Pro.

Figure 9.27 Hilite template.

5. With the hilite.png image file active, choose Edit, Copy. You must make sure that you have no selections in the image. You can make doubly sure by choosing the Selection tool and right-clicking inside the image; this will deselect any selections that might exist—sometimes they can be so small that you don't notice them.
6. Find your working copy of player.psp in Paint Shop Pro, and choose Edit, Paste, Paste as New Layer. This will paste the hilite image you copied into a new raster layer. Rename the layer as "Hilite". The first thing you will probably notice is that it is larger than the head and neck template area.

7. Choose Image, Resize and you will get the Resize dialog box. Clear the Resize All Layers box, and then in the Pixel Dimensions frame, set Height and Width to 80, and choose Percent from the combo box to the right. Click OK.
8. Now lower the opacity of the new layer to about 80 percent.
9. Make the skin layer invisible.
10. Drag the image around until you get the best fit over the UV template of the head and neck.

    You should get an image that looks like Figure 9.28.

    Being the astute observer that I know you are, you've no doubt noticed that although the hilite template fits quite well at 80 percent of its full size, it's not exactly right. For one thing, the eyes are wrong—the hilite of the eye area needs to be slanted to match the contours of the UV triangles. We're going to fix that right now.

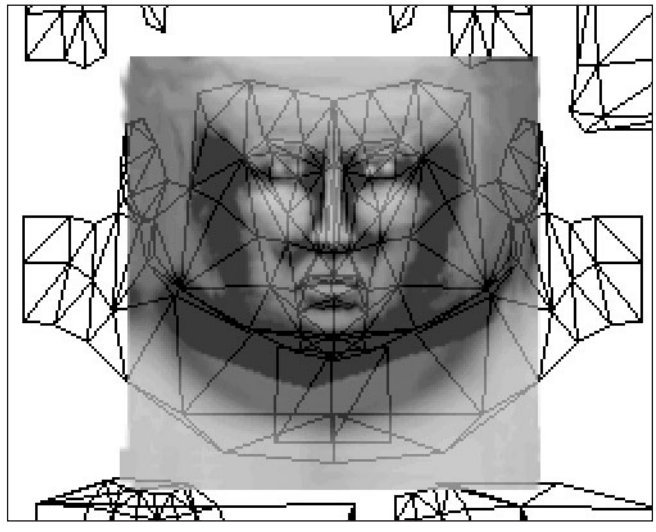


Figure 9.28 Hilite template applied over the head and neck UV template.

11. Make sure that the new raster layer with the hilites on it is active by making sure that its entry in the Layer palette is selected. Then choose the Selection tool, which is the fifth tool from the top of the Tool palette.
12. Select an area around the right that encompasses the eye, the brow above, and a small amount of the upper cheek below, but no part of the bridge of the nose.
13. Choose Image, Rotate, Free Rotate. In the Free Rotate dialog box, click on Left and then click on Free. Type **5.00** into the Free box and click OK.

tip

The hilite.png template was created by taking several full-face photos and drawings and stretching the contrast of each quite a bit in grayscale. The images were then all overlaid and averaged to give a resulting template. That result was then tested in a few models and manually tweaked a few times. The originals were chosen to be all of the same face shape and type. Different templates for different ethnicities and face shapes were made this way.

14. The eye area has rotated. Press Ctrl+Up Arrow to nudge the selection up once and then Ctrl+Right Arrow to nudge it once to the right.
15. Right-click in the image to deselect the selection, and check to make sure that it is aligned correctly. If it isn't, press Ctrl+Z and you will get your selection back. You can then rotate it or nudge it some more.
16. When you have the eye hilites placed, repeat this process for the other eye.
17. Lower the opacity for the hilite layer to about 20 percent, and raise the opacity of the skin layer to 100 percent.
18. Save your file (just a precaution).

Now you have something like Figure 9.29, with the rough shading and coloring of flesh tones showing the major features of the face. At this point it becomes a case of filling in the details. You can go ahead and do it however you like. Zoom in close and use the Air Brush and Paint Brush tools. Add lip color, eyebrows, eye details, and ears. You might find it hard to put actual eye color in, but give it a try.

Figure 9.29 Hilite template applied over the skin layer.

Eye detail can be added by creating ellipse objects on a scratch vector layer, sizing and rotating them correctly, and placing them over the eye areas. When you create eyes remember that the colored area in the center, the combined pupil and iris regions, usually has a white or otherwise light spot offset a bit to one side and a bit above center, as shown in Figure 9.30.

Also remember that certain areas of the face are usually lighter in tone than others, like the upper part of the lower lip, the upper eyelid, the nostrils, and so on.

You can make a good five-day stubble by using the Paint Brush with the density set low, like 25 percent or so. Dab the brush in the areas where it is needed. Make a moustache by applying the stubble brush over and over to the upper lip. Encourage yourself to experiment!

Eventually you will end up with something like Figure 9.31.

Figure 9.30 An eye.

Hair and Hands

Next we'll tackle the hair and hands of the Standard Male. We'll do these two together because they both use skin (flesh) tones (the guy is going to have a bald spot). Once these are done, we are finished with the skin part of the skin. Or something like that.

Both of the next subsections will be using the skin layer in addition to other layers.

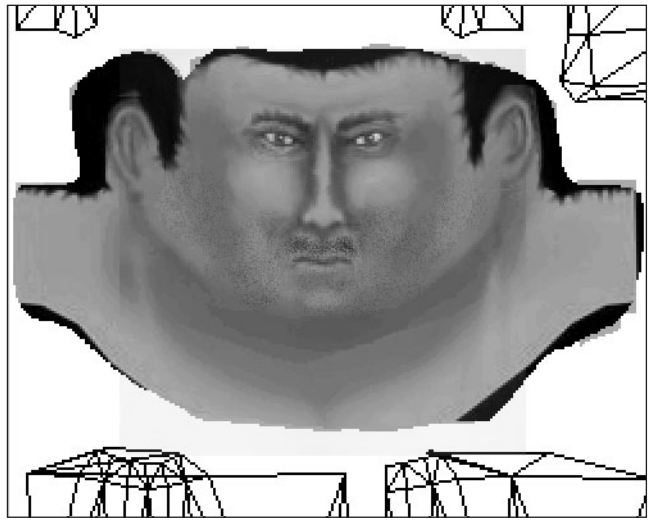


Figure 9.31 Finished face and neck.

Hair Textures

Hair has a pattern, though not a specific pattern. There is often quite a bit of randomness, but nonetheless there is a grain, if you will, like the grain in a wooden plank or the lay of a lawn. There's a clue there!

Try this:

1. Locate the hair portion of the UV template in your working file, player.psp.
2. Draw an object that encompasses the hair, and set the fill to match the color of the hair you used in the bits that show in the head area as in Figure 9.32.
3. Copy that object and paste the copy into another new layer. Modify the fill of that object. (Reminder: You can do this by double-clicking on the object with the Object Select tool and then clicking on the Fill color box.)
4. Set the hair RGB color value to those listed in Table 9.5.
5. Select the Texture check box.
6. Click the Current Texture display box and select Grass02 from the list that pops up.
7. Set Angle to 90 and Scale to 50.

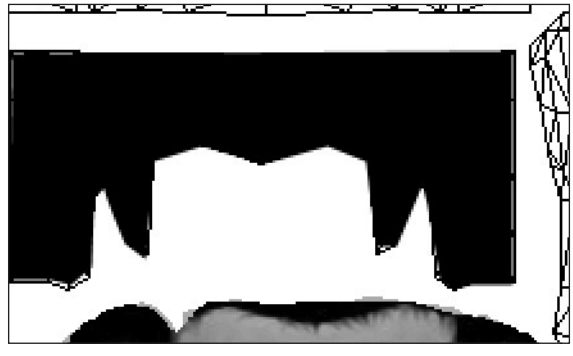


Figure 9.32 Filled hair template area.

Table 9.5 Hair Color RGB Settings

Color Component	Value
Red	251
Green	178
Blue	129

8. Click OK to close the Color dialog box and then again to close the Properties dialog box. You will get something similar to Figure 9.33.
9. Merge the new layers you created with your skin layer, using the Merge Visible technique I showed you earlier.
10. After you do this, the layer will be named "Merged". Rename it to "Skin" again, by right-clicking on the layer's name in the Layer palette, choosing Rename, and then typing in the name.

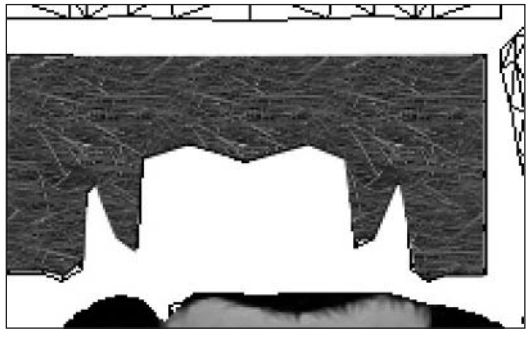


Figure 9.33 Textured hair.

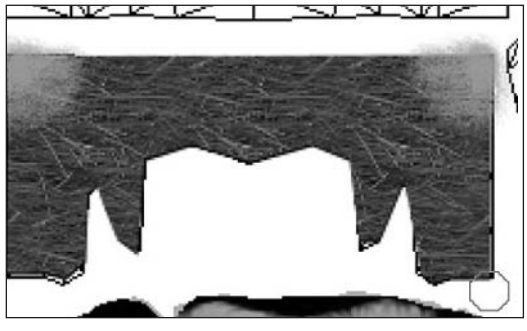


Figure 9.34 The font of wisdom under construction—the bald spot.

11. Now for the bald spot. If you look at how the triangles in the UV template are arranged you can see that the upper-left corner of the hair area and the upper-right corner of the hair area meet when they are wrapped back onto the model. The place where they meet is the crown of the head, which just so happens to be one of the two places where classic male-pattern baldness begins!

    Choose the Air Brush and set its size to about 32, its density to about 25, and its foreground color to the highlight flesh tone found in Table 9.4.
12. In each of the corners, spray some bald skin on, sparser toward the inner areas and denser as you move toward the corners, until you have a substantial patch of bare skin and a surrounding area of varying thinness (see Figure 9.34). Don't worry about overspraying the edges, those areas outside are not going to be rendered.

The Hands

The hands need to be skinned on three sides. You should use the basic flesh tone, with some shadow color for areas between the fingers.

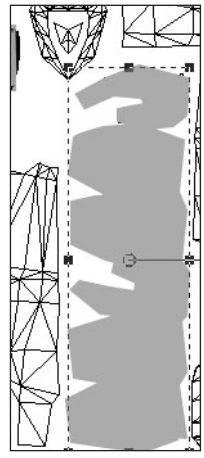


Figure 9.35 Hand area.

1. Once again using the Pen tool with the Point to Point Segment Type, draw an object that surrounds the area that comprises the hand UV template (Figure 9.35).
2. Set the fill color of the object you just made to the basic flesh tone.
3. Start a new vector layer.
4. Using the Pen tool with the Segment Type set to Freehand, Line Width set to 2.0, and Color set to black, draw the lines that separate the fingers. Use Figure 9.36 as a guide.
5. Put the Pen tool back into Point to Point mode, and draw a fingernail. Make sure the line color is black, and use a fairly bright pink for the fill color.
6. Place your lines and fingernails appropriately (as in Figure 9.36), and fiddle with the shapes until you are happy.
7. Set the opacity of the layer to about 10 or so. That bright pink fingernail color is not so bright anymore.
8. Merge the two layers you just created into the skin layer.
9. Using the touchup brushes (ninth from the bottom of the Tools toolbar) and the Air Brush tool, add shading and irregularity to the lines as in Figure 9.37.
10. Weaken some of the darker lines. Add lighter highlights around the main knuckles and darker wrinkles around the other knuckles.
11. Eventually you will arrive at something that works for you, similar to Figure 9.38.

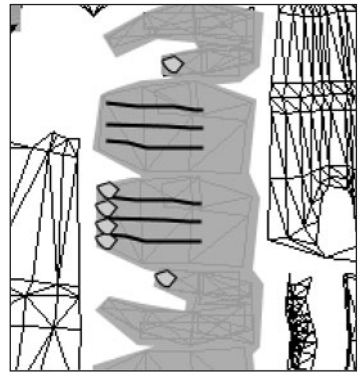


Figure 9.36 Finger lines and fingernails.

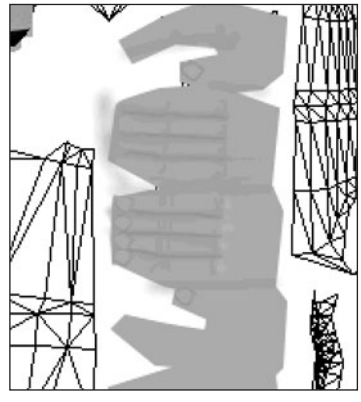


Figure 9.37 Adding hand details.

The Clothes

We'll spend most of our time remaining in this chapter working on the jacket. You've already learned and applied almost all of the new skills required to do the clothing.

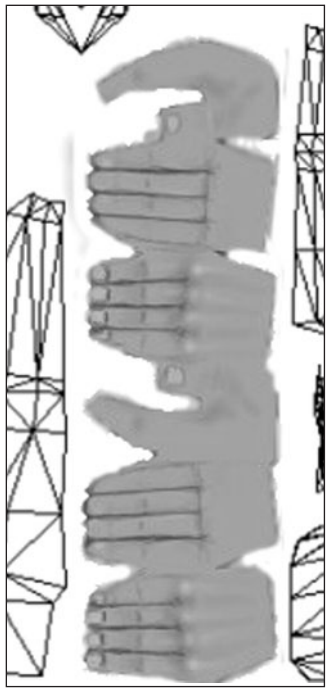


Figure 9.38 The finished hands.

The Jacket

It's a leather jacket. Quite a nice one too. Wouldn't mind one like that myself! The color is a basic brown, with the usual darker shadows and lighter highlights, just like with the flesh tones. Things to note are that the jacket "blouses" at the waist and at the cuffs. This is a wrinkling effect that occurs as the material is gathered in for the seam work in those areas.

1. Start off by drawing objects around the back, the front, the waist, the cuffs, the collar, and the sleeves in a fashion similar to what we've done in the past (see Figure 9.39). Make sure you do this on a new layer and name it "Jacket".
2. Set the fill color to the basic brown, using the values shown in Table 9.6.
3. On the Tool Options palette, just to the right of the Presets box, is another box with brush configuration choices. Click on that box, and then choose Small Bristles Hard from the icon list.

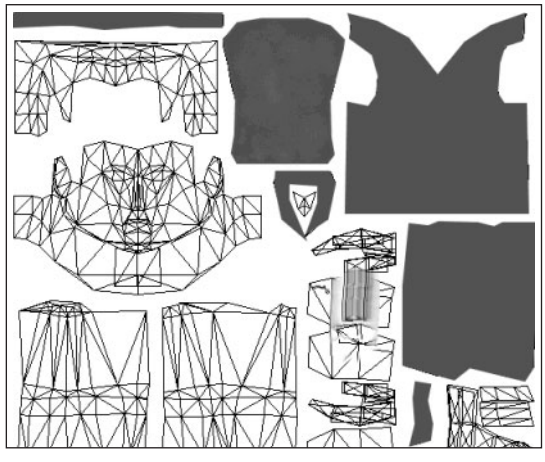


Figure 9.39 The jacket pieces.

Table 9.6 Jacket Color RGB Settings

Color Component	Value
Red	140
Green	68
Blue	62

4. To get that stippled leathery look, choose the Air Brush and set it as shown in Table 9.7.

Table 9.7 Air Brush Settings

Setting	Value
Shape	Round
Size	26
Hardness	100
Step	10
Density	50
Thickness	100
Rotation	0
Opacity	90
Blend Mode	Normal
Rate	5

5. Merge all your new layers onto the skin layer.
6. Spray the leather areas of the jacket with short sharp strokes—just enough to get the stippled look to appear. Do this for all the leather areas: back, front, collar, and sleeves. Figure 9.40 gives an idea what I've done: The back (on the left) has the stippled look, while the visible part of the front (on the right) does not.
7. Use the Lighten/Darken brush (the tenth brush down in the Tool palette) to make the contours of the gathers at the bottom of the front of the jacket.
8. Use the Smudge brush and the other touchup brushes to tweak the contours to your liking (for example, as in Figure 9.41).
9. You can create the zipper and the zipper flap by using the Pen tool to draw a line from the neck to the bottom. Make one line with a width of about 3.0, and then copy it and paste the copy next to the original line.
10. Edit the properties of the new line and change its width to 7. This will be the flap.
11. Merge the new layers to the skin layer, and then touch up the zipper area with stippling and make other tweaks to get it to coordinate with the other areas of the jacket.
12. You can do all the other areas of the jacket in the same way as shown in steps 4 to 11.

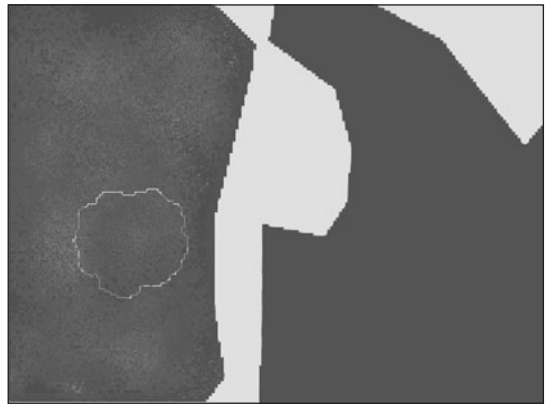


Figure 9.40 Getting close to that leathery look.

Team LRN

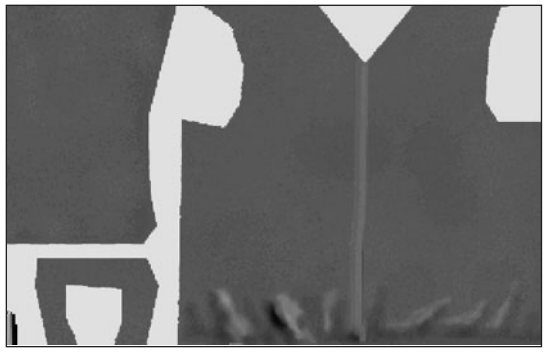


Figure 9.41 That leathery look.

The Trousers

The trousers can be done using exactly the same techniques as used for the jacket. You just need to use different colors and perhaps a different texture or airbrush density or step value. By now, you should be pretty handy with the toolkit in Paint Shop Pro, so I'll leave you to do the trousers on your own. Don't forget to make a belt—it goes at the bottom of the trouser area in the UV template.

The Boots

The final area to apply texture to is the boot area. Again, you've practiced all the techniques required to make the boots as well. There is one thing I want to show you, though, that will help, and that is the built-in textures in the Color dialog boxes.

If you click on the Color box in the Materials palette, or the Color box in the Properties dialog box of an object, you will see a tab for Patterns, and then to the right side, a box for Textures. The textures will be applied to whatever color mode (tab) you have selected, so that you can have, say, textures applied to patterns. In the Pattern list there is a Tire Tread pattern that would be suitable for the bottom of the heel of a boot, and in the Textures list there are many textures that would be suitable for different parts of the boot.

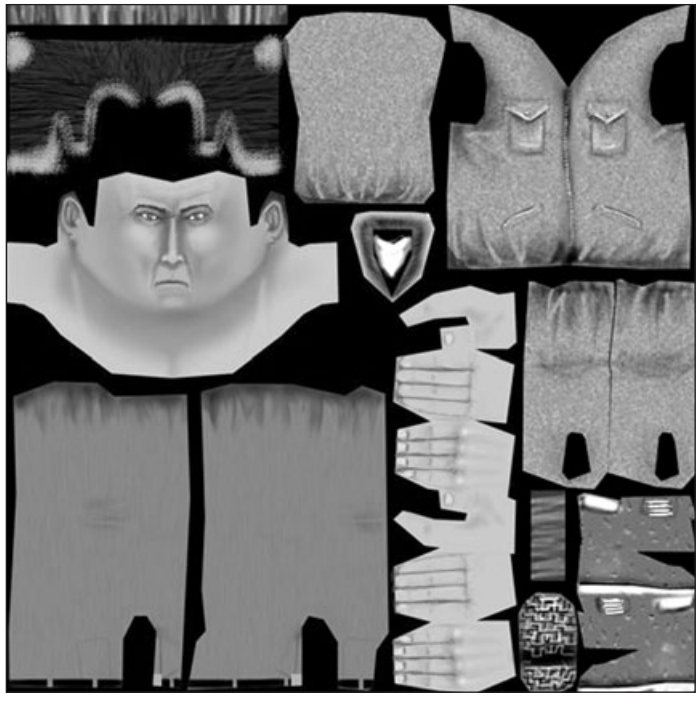


Figure 9.42 Standard Male skin.

Make sure you save your work in player.psp, and then save another version as C:\3DGPAi1\resources\ch9\player.jpg.

Figure 9.42 shows the complete skin for the Standard Male.

Trying It On for Size

As you learned earlier in the chapter, you can use the Show Book Models shortcut and load the player.dts model. You will be able to view the Standard Male character with your new skin on it. You'll probably see areas that need fixing up, and so go ahead and do just that.

Moving Right Along

In this chapter you learned how UV unwrapping relates to the texture files known as *skins*. And you learned how to apply that understanding to images for game objects ranging from the simple (a soup can) to the complex (a human character).

I hope you also take away from the chapter the idea that hand-drawn concept artwork is a useful tool. Draw everything in sketch form before you start working on your models. It's a great help.

Finally, you can see that a fully featured image processing tool like Paint Shop Pro has quite a few features to ease the effort of creating images for skins. We've only scratched the surface of what the program can do. Don't be shy about using Paint Shop Pro's built-in Help utility. It's well done and chock-a-block-full of information.

If you want to make great skins, you are going to need to practice, practice, and practice some more. Here are some of the many ways to do this:

- Create your own models and make the skins.
- Make skins for other people's models.
- Make skins for other people for popular games like *Half-Life* and *Tribes*.
- Make monster skins, policeman skins, airplane skins, light pole skins.
- Make a set of stock skins.
- Make skin templates that you can use to make the skinning task easier.

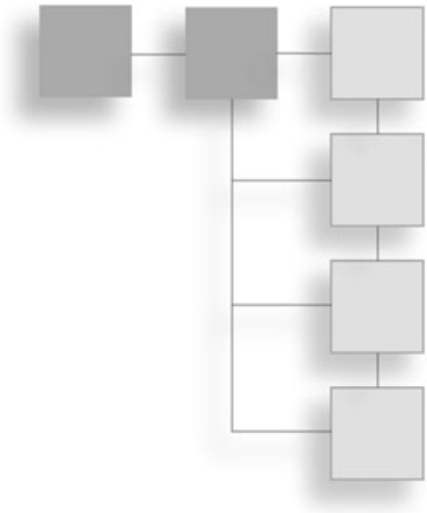
But most of all, get down and do it!

In the next chapter, we will continue with the visual aspects of developing our game, but this time we will be looking at how to create graphical user interface (GUI) elements, by using Torque script to insert images and controls.

This page intentionally left blank

Team LRN

CHAPTER 10



CREATING GUI ELEMENTS

As you've seen by now, there is more to a 3D game than just the imaginary world into which the player plunks his avatar. There is the real need to provide the player with some method to make selections and otherwise control the game activities. Generally, we provide a *Graphical User Interface* (GUI) to the player to interact with the program. The menu we employed at the start-up of the program, where the player clicks on buttons to launch the game, change the setup, or quit; the dialog box that shows the client's loading progress; the dialog box that asks if the player really wants to quit—these screens are all examples of GUIs.

If you take a look at Figure 10.1, you can see a sample of the variety of elements found within these interface screens.

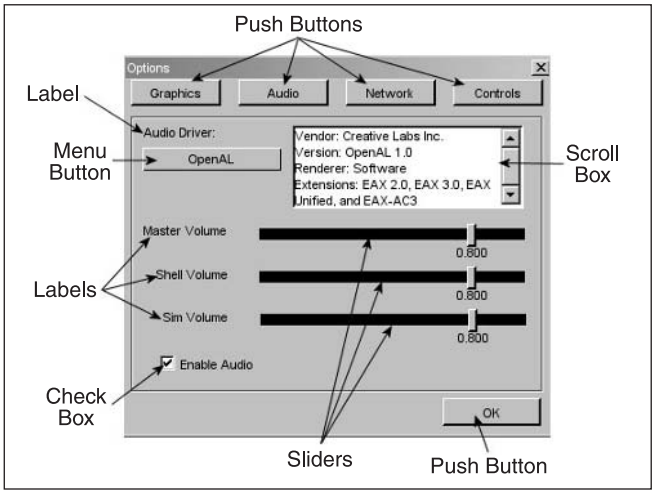


Figure 10.1 Common graphical user interface elements.

Some of the elements are things we can interact with:

- push buttons
- radio buttons
- edit boxes
- check boxes
- menus
- sliders

Team LRN

Some of the elements are things we can just look at:

- frames
- labels
- backgrounds

Also, during the course of discussions about graphical user interfaces, you may find the terms *GUI*, *window*, *interface*, and *screen* used interchangeably. I'll stick to the words *interface* and *screen* as much as possible, although contextually it might make more sense to use *GUI* or *window* from time to time. *GUI* is best used to describe the entire game interface with the player as a whole. *Window* is a term that most people tend to associate with the operating system of their computer.

The names of GUI items that are available by default with Torque don't differentiate between whether they are interactive and noninteractive GUI elements.

If you are familiar with X-Windows or Motif, you will probably have encountered the term *widgets*. If so, your definition of widgets may be a fair bit broader than the one I am about to use here. In our situation, widgets are simply visual portions of a displayed GUI control. They convey information or provide an aesthetic appearance and offer access to defined subcontrol elements.

For example, Figure 10.2 portrays a scroll bar. Within the scroll bar are the *thumb*, *arrow*, and *bar* widgets. These aren't controls in their own right but rather are necessary specialized components of the control to which they belong.

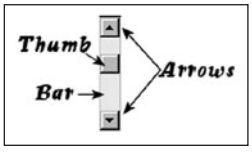


Figure 10.2 Scroll bar widgets.

It is possible for a control to use another control as a widget. In fact, every control in a screen can be considered a widget within the control that defines the screen. This will become clearer later on. I will only use the term *widget* to refer to a specialized component of a control that is not itself a control.

Worth noting is the fact that you can create your own GUI elements using Torque Script if the ones that are available by default don't suit your needs.

Controls

The name says it all—*controls* are graphical items provided to the program user to control what the program will do. In Torque, interactive controls are used by clicking on them or click-dragging the mouse across them. Some controls, like edit boxes, also require you to type in some text from the keyboard. Some of the controls have built-in labels that identify their purpose, and some will require you to create an accompanying noninteractive control to provide a label. Noninteractive controls, as the name implies, are used to only display information and not to capture user input.

Team LRN

Torque provides a number of default controls right out of the box; the most commonly used ones are listed next. You will have encountered a few of these controls in earlier chapters, and we will discuss several more of them in this chapter. You can use them as is; you can modify them by adjusting the control's profile; or you can use them as the basis for defining new controls.

```
GuiArrayCtrl                    GuiControlListPopUp          GuiPlayerView
GuiAviBitmapCtrlGuiBackgroundCtrl GuiCrossHairHud            GuiPopUpBackgroundCtrl
GuiBitmapBorderCtrl             GuiEditCtrl                  GuiPopUpMenuCtrl
GuiBitmapButtonCtrl             GuiFadeinBitmapCtrl          GuiPopUpTextListCtrl
GuiBitmapButtonTextCtrl         GuiFilterCtrl                GuiProgressCtrl
GuiBitmapCtrl                   GuiFrameSetCtrl              GuiRadioCtrl
GuiBorderButtonCtrl             GuiHealthBarHud              GuiScrollCtrl
GuiBubbleTextCtrl               GuiInputCtrl                 GuiShapeNameHud
GuiButtonBaseCtrl               GuiInspector                 GuiSliderCtrl
GuiButtonCtrl                   GuiMenuBackgroundCtrl        GuiSpeedometerHud
GuiCanvas                       GuiMenuBar                   GuiTerrPreviewCtrl
GuiCheckBoxCtrl                 GuiMenuTextListCtrl          GuiTextCtrl
GuiChunkedBitmapCtrl            GuiMessageVectorCtrl         GuiTextEditCtrl
GuiClockHud                     GuiMLTextCtrl                GuiTextEditSliderCtrl
GuiConsole                      GuiMLTextEditCtrl            GuiTextListCtrl
GuiConsoleEditCtrl              GuiMouseEventCtrl            GuiTreeViewCtrl
GuiConsoleTextCtrl              GuiNoMouseCtrl               GuiWindowCtrl
```

Figure 10.3 shows a screen used to select missions to play. There is a list of available missions on the client, some buttons to run the mission or go back to the main menu, and a check box to indicate whether you want to host this mission for other players. Note, too, that there is a background, which is the same as the background used for our Emaga game program's start-up menu.

What we'll do is examine each of the screen's GUI elements in detail.

GuiChunkedBitmapCtrl

The GuiChunkedBitmapCtrl class is usually used for the large backgrounds of interfaces, like menu screens. Figure 10.4 shows such a background. The name derives from the concept of breaking up an image into a collection of smaller ones (chunked bitmaps) in order to improve display performance.

Here is an example of a GuiChunkedBitmapCtrl definition:

```
new GuiChunkedBitmapCtrl(MenuScreen) {
   profile = "GuiContentProfile";
   horizSizing = "width";
   vertSizing = "height";
```

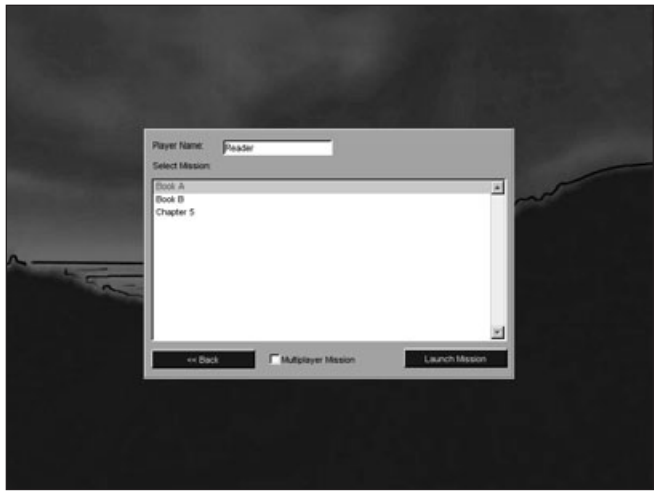


Figure 10.3 Start mission interface screen.

Figure 10.4 GuiChunkedBitmapCtrl background sample.

```
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
   bitmap =
"./interfaces/emaga_back-
ground";
   // insert other controls
here
};
```

The first thing to note about this definition is the line "// insert other controls here". Typically, a GuiChunkedBitmapCtrl control would contain other controls, functioning as a sort of super-container. All other controls in a given screen using this control would be children, or subelements, of this control. This line is a comment, so in and of itself, it has no effect on the control's definition. I include it here to indicate where you would start nesting other controls.

Note the `extent` property, which specifies a width of 640 and a height of 480. These are "virtual pixels" in a way. Any subelements you insert in this control will have a maximum area of 640×480 to work with for positioning and sizing. These virtual pixels are scaled in size according to the actual canvas size, which you can change by setting the value of the global variable `$pref::Video::windowedRes` and then calling `CreateCanvas`, or if you already have a canvas, calling `Canvas.Repaint`;—we used `CreateCanvas` in Chapter 7.

The `minExtent` property specifies the smallest size that you will allow this control to be shrunk down to when using the Torque built-in GUI Editor. We will use that editor later in this chapter.

GuiControl

The GuiControl class, as shown in Figure 10.5, is a sort of generic control container. It's often used as a tablike container, or as what other systems often call a *frame*. With it, you can gather together a collection of other controls and then manipulate them as a group.

Here is an example of a GuiControl definition:

```
new GuiControl(InfoTab) {
   profile = "GuiDefaultProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
};
```

Probably the property you will be most interested in is the `visible` property. You will probably want to programmatically make the control visible or invisible based on the contents (the other controls) you place within the control. You can do that this way:

```
InfoTab.visible = true;
InfoTab.visible = false;
```

Note that `true` is the same as 1 or "1" and `false` is the same as 0 or "0".

GuiTextCtrl

The GuiTextCtrl, as shown in Figure 10.6, is a straightforward, commonly used control. You can use it to display any text you want. You can put it on an interface with no text and then fill in the text as the game progresses.

Here is an example of a GuiTextCtrl definition:

```
new GuiTextCtrl(PlayerNameLabel) {
   profile = "GuiTextProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "183 5";
   extent = "63 18";
```

Figure 10.5 GuiControl sample.

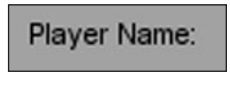


Figure 10.6 GuiTextCtrl sample.

```
   minExtent = "8 8";
   visible = "1";
   text = "Player Name:";
   maxLength = "255";
};
```

You would specify the text font and other characteristics with your choice of `profile`. You can change the contents quite easily in this example by doing the following:

```
PlayerNameLabel.text = "Some Other Text";
```

The `maxLength` property allows you to limit the number of characters that will be stored with the control. Specifying fewer characters saves memory.

GuiButtonCtrl

The GuiButtonCtrl, as shown in Figure 10.7, is another clickable control class. Unlike GuiCheckBoxCtrl or GuiRadioCtrl, this class does not retain any state. Its use is normally as a *command interface control*, where the user clicks on it with the expectation that some action will be immediately invoked.

Here is an example of a `GuiButtonCtrl` definition:

```
new GuiButtonCtrl() {
   profile = "GuiButtonProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "16 253";
   extent = "127 23";
   minExtent = "8 8";
   visible = "1";
   command = "Canvas.getContent().Close();";
   text = "Close";
   groupNum = "-1";
   buttonType = "PushButton";
};
```

The most significant property is the `command` property. It contains a script statement to be executed when the button is pressed. This example will close the interface screen being shown in the canvas.

Figure 10.7 GuiButtonCtrl sample.

Another feature is the `buttonType` property. This can be one of the following:

- `ButtonTypePush`
- `ButtonTypeCheck`
- `ButtonTypeRadio`

The property `groupNum` is used when the `buttonType` is specified to be `ButtonTypeRadio`. Radio buttons in an interface screen that have the same `groupNum` value are used in an exclusive manner. Only the most recently pressed radio button will be set to the checked value (true); all others in the group will be unchecked. Otherwise, the radio button type works the same as the GuiCheckBoxCtrl class, described in the next section.

This control is also used as a base for deriving the three button classes shown previously. You would probably be better off to use the specialized classes GuiCheckBoxCtrl and GuiRadioCtrl for types `ButtonTypeCheck` and `ButtonTypeRadio`, rather than this control, because they have additional properties.

So the upshot is, if you use this control, it will probably be as a `ButtonTypePush`.

GuiCheckBoxCtrl

The GuiCheckBoxCtrl, as shown in Figure 10.8, is a specialized derivation of the GuiButtonCtrl that saves its current state value. It's analogous to a light switch or, more properly, a locking push button. If the box is empty when you click the control, the box will then display a check box. If it is checked, it will clear the check mark out of the box when you click the control.

Here is an example of a GuiCheckBoxCtrl definition:

```
new GuiCheckBoxCtrl(IsMultiplayer) {
   profile = "GuiCheckBoxProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "155 272";
   extent = "147 23";
   minExtent = "8 8";
   visible = "1";
   variable = "Pref::HostMultiPlayer";
   text = "Host Mission";
   maxLength = "255";
};
```

If you specify the `variable` property, then the value of the specified variable will be set to whatever the current state of the control is after you've clicked it.

Figure 10.8 GuiCheckBoxCtrl sample.

When the control is first displayed, it will set its state according to the value in the specified variable. You need to make sure that the variable you use contains appropriate data.

You can also specify the text label that will be displayed next to the check box using the `text` property.

Note that the GuiRadioCtrl control functions much like this control, except that it automatically enforces the principle that only one button in the same group will be checked.

GuiScrollCtrl

The GuiScrollCtrl class, as shown in Figure 10.9, is used for those famous scrolling lists that everyone likes. Okay, so not everyone may like them, but everyone *has* used them.

Here is an example of a GuiScrollCtrl definition:

```
new GuiScrollCtrl() {
   profile = "GuiScrollProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "14 55";
   extent = "580 190";
   minExtent = "8 8";
   visible = "1";
   willFirstRespond = "1";
   hScrollBar = "dynamic";
   vScrollBar = "alwaysOn";
   constantThumbHeight = "0";
   childMargin = "0 0";
   defaultLineHeight = "15";
   // insert listing control here
};
```

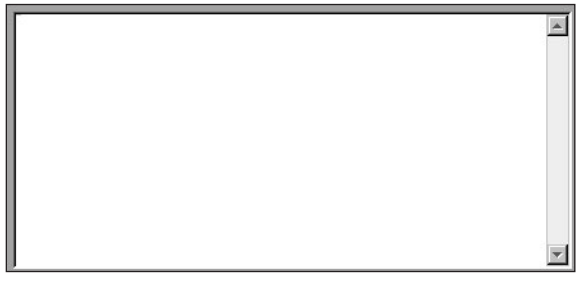


Figure 10.9 GuiScrollCtrl sample.

Normally we would populate a scroll control with a list, usually defined by the contents of a GuiTextListCtrl control. The control containing the list would be added as a subelement of this control.

The `willFirstRespond` property is used to indicate whether we want this control to respond to arrow keys

when they are pressed (to control scrolling) or to let other controls have access to arrow key inputs first.

Both the `hScrollBar` and `vScrollBar` properties—referring to the horizontal and vertical bars, respectively—can be set to one of these modes:

- **alwaysOn.** The scroll bar is always visible.
- **alwaysOff.** The scroll bar is never visible.
- **dynamic.** The scroll bar is visible only when the list exceeds the display space.

The property `constantThumbHeight` indicates whether the *thumb*, the small rectangular widget in the scroll bar that moves as you scroll, will have a size that is proportional to the number of entries in the list (the longer the list, the smaller the thumb) or will have a constant size. Setting this property to 1 ensures a constant size; 0 will ensure proportional sizing.

The property `defaultLineHeight` defines in virtual pixels how high each line of the control's contents would be. This value is used to determine how much to scroll when a vertical arrow is clicked, for example.

Finally, `childMargin` is used to constrain the viewable space inside the parent control that would be occupied by whatever control contained the list to be scrolled. In effect, it creates a margin inside the scroll control that restricts placement of the scroll list. The first value is the horizontal margin (for both left and right), and the second is the vertical margin (both top and bottom together).

GuiTextListCtrl

The `GuiTextListCtrl`, as shown in Figure 10.10, is used to display 2D arrays of text values.

Here is an example of a `GuiTextListCtrl` definition:

```
new GuiTextListCtrl(MasterServerList) {
   profile = "GuiTextArrayProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "2 2";
   extent = "558 48";
   minExtent = "8 8";
   visible = "1";
   enumerate = "0";
   resizeCell = "1";
   columns = "0 30 200 240 280 400";
   fitParentWidth = "1";
   clipColumnText = "0";
   noDuplicates = "false";
};
```

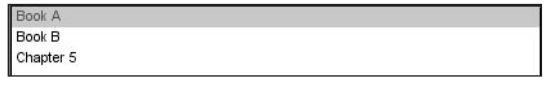


Figure 10.10 `GuiTextListCtrl` sample.

The `enumerate` property indicates which line of text is presented as highlighted.

You can allow the cells to be resized with the GUI Editor by setting the `resizeCell` property to `true`.

Each record, or line, in the array has space-delimited fields. You can format the display of these fields by using the `columns` property to indicate at which column number each field will be displayed.

The `fitParentWidth` property indicates whether the control will be enlarged in size to fill the available display space of any control that might contain this control.

We can decide whether overlong text in each column is to be clipped, or will be left to overrun adjoining columns, by setting the `clipColumnText` property.

We can automatically prevent the display of duplicate record entries by setting the `noDuplicates` property to `true`.

GuiTextEditCtrl

The `GuiTextEditCtrl`, as shown in Figure 10.11, provides a tool for users to manually enter text strings.

Here is an example of a `GuiTextEditCtrl` definition:

```
new GuiTextEditCtrl() {
   profile = "GuiTextEditProfile";
   horizSizing = "right";
   vertSizing = "bottom";
   position = "250 5";
   extent = "134 18";
   minExtent = "8 8";
   visible = "1";
   variable = "Pref::Player::Name";
   maxLength = "255";
   historySize = "5";
   password = "0";
   sinkAllKeyEvents = "0";
   helpTag = "0";
};
```

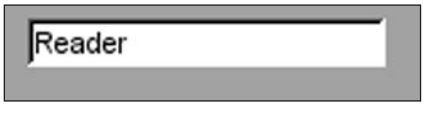


Figure 10.11 `GuiTextEditCtrl` sample.

With this control, the `variable` property is the key one. When the user types a string of text into the control's edit box, that string is entered into the variable indicated. When the control is first displayed, the contents of the indicated variable are stuffed into the edit box for display.

Text edit controls have a nifty history feature that can be quite handy. All of the previous entries—up to a maximum specified by `historySize`—are saved and can be recalled using the Up Arrow key to go back in history or the Down Arrow key to go forward.

If you are using this control to accept a password, then set the `password` property to `true`. The control will substitute asterisks ("*") in place of whatever is typed by the user so that bystanders can't see what is being typed.

The `sinkAllKeyEvents` property, when set to `true`, causes the control to throw away any keystrokes that it receives but doesn't understand how to handle. When `sinkAllKeyEvents` is set to `false`, these keystrokes will be passed to the parent.

The Torque GUI Editor

Torque has an editor built in for creating and tweaking interfaces. You can invoke the GUI Editor by pressing the F10 key (this is defined in the common code base scripts, but you can change it if you want). You are perfectly free to ship your game with this editor code, or you can remove it in any shipping version to ensure that people will not fiddle with the interfaces. Or you can modify it to suit your heart's desire!

The Cook's Tour of the Editor

When you launch the editor by pressing the F10 key, the editor will appear and load whatever interface is current, making it ready for editing.

Visually, there are four components to the GUI Editor: the Content Editor, the Control Tree, the Control Inspector, and the Tool Bar. Figure 10.12 shows the GUI Editor open and working with one of the earlier main menu screens from the Emaga sample game.

Figure 10.12 The Torque GUI Editor.

The Content Editor

The Content Editor is where you can place, move, and resize controls. In Figure 10.12 the Content Editor is the large rectangular area at the upper left in the GUI Editor view.

Selection

Normally, you select a control by clicking the mouse on it. Some controls can be difficult to select because of their

positions. Another way to select controls is by using the Control Tree, which is covered in a later section.

If you hold down the Shift key while clicking the mouse (*shift-clicking*) on several controls, you can select more than one control at once. Each time you shift-click you add that control to the selection. The sizing knobs turn white and can no longer be used to size the control. You can still move the controls. Only controls that share the same parent can be selected at the same time.

Movement

Move a control by clicking and dragging its content area after selecting it. When you move controls, be aware of which controls they may be contained by—when you drag the control to one side or another, you may be dragging it outside the display area of its parent control, and you don't want that.

Resizing

You resize a control after selection by clicking on and dragging one of the eight black sizing knobs. As with movement, you need to stay aware of how the control you are resizing is related to other controls. The sizing might be restricted by a parent control's display area. Figure 10.12 shows the sizing knobs, attached to the Start Mission button.

Adding

The parent control of the currently selected control is outlined with a yellow and blue band. This control is known as the *Current Add Parent*. Any new control created from the toolbar or pasted from the Clipboard will be added to this control. The Current Add Parent control can be set manually by either clicking one of its children or right-clicking the control itself.

The Control Tree

The Control Tree shows the current content control hierarchy. It is in the upper-right corner of the GUI Editor view.

Parent controls, also called *containers*—controls that contain other controls—have a little box to the left of their entry in the tree. If the box is a plus sign, clicking it will expand that control into the list, bringing the child controls into view. If you click it when it looks like a minus sign, it will contract the control's list back to a single entry comprising solely the parent control.

Clicking any control in the tree will cause it to be selected in the Content Editor view and cause the control's properties to be displayed in the Control Inspector view. You can see this effect by looking back at Figure 10.12.

The Control Inspector

The Control Inspector is where the currently selected control's attributes are displayed. It is located in the lower-right corner of the GUI Editor, below the Control Tree. All of the properties of a control are displayed in the Inspector and can be changed here. After changing a value, you must click the Apply button to assert the changes.

When first displayed, all of the properties are collapsed visually within categories, such as Parent, Misc, and Dynamic Fields. To access the properties within those categories, simply click the buttons in the Inspector view that have those category names, and the list expands, giving you edit boxes and buttons with which you can manipulate the properties.

The Tool Bar

The Tool Bar contains functions for creating new controls and for aligning and spacing them. It has several command buttons that operate on the current selection set, and create and save GUIs. The Tool Bar also has pop-up menus for creating new controls and changing the currently edited GUI. The functions of the buttons are described in Table 10.1.

Table 10.1 Tool Bar Button Functions

Button	Description
Align Left	Aligns the left edge of all selected controls to the leftmost point of all the selected controls.
Align Right	Aligns the right edge of all selected controls to the rightmost point of all the selected controls.
Center Horiz	Horizontally centers all selected controls in the rectangle that bounds all the selected controls.
Align Top	Aligns the top edge of all selected controls to the topmost point of all the selected controls.
Align Bottom	Aligns the bottom edge of all selected controls to the bottommost point of all the selected controls.
Help	Displays the Help dialog box.
Space Vert	Vertically spaces all selected controls evenly.
Space Horiz	Horizontally spaces all selected controls evenly.
Bring Front	Arranges the selected control in front of its sibling controls.
Send Back	Arranges the selected control behind its sibling controls.
New	Brings up a dialog box with which the user can create and name a new control for editing.
Save	Brings up a dialog box with which the user can save the current interface to a file.
New Control (pop-up)	Displays a list of all controls from which the user can select one to add to the current content control.
Show GUI (pop-up)	Displays the name of the interface (GUI) currently being edited. Selecting this pop-up allows the user to choose a screen to edit from all loaded interfaces.

Team LRN

Keyboard Commands

In addition to using mouse selection and GUI button clicks, the user has a number of keyboard commands available. Table 10.2 shows these commands.

Table 10.2 GUI Editor Keyboard Commands

Keys	Name	Description
Ctrl+A	Select All	Selects all the controls in the Current Add Parent.
Ctrl+C	Copy	Copies the currently selected control(s) to the Clipboard.
Ctrl+X	Cut	Cuts the currently selected control(s) to the Clipboard.
Ctrl+V	Paste	Pastes any control on the Clipboard into the Current Add Parent.
Arrow Keys	Movement	Moves the current control selection 1 pixel in the direction of the arrow.
Shift+Arrow Keys	Movement	Moves the current control selection 10 pixels in the direction of the arrow.
Delete/Backspace	Delete	Deletes the current control selection.

Creating an Interface

In this section, you will see how easy it is to create and employ an interface using the Torque GUI Editor.

You should note that the Torque GUI Editor assumes your screen resolution is set to a minimum resolution of 800×600. You may find it more useful to use a higher resolution, to allow the different views more room to display their data.

1. Using Windows Explorer, browse into the C:\3DGPAi1 folder and then double-click on the Run fps Demo shortcut.
2. When the GarageGames/Torque menu screen appears, press the F10 key.

   The editor controls will appear on the bottom and right side of the screen and enable you to immediately start editing the screen you were previously viewing.
3. Click the New button and enter a name for the new interface—do not use spaces in the name. Use "MyFirstInterface" for the name.
4. Leave the class as GuiControl, and then press the Create button. You will now have a nice new interface to work with.
5. In the Tree view, select the control named "MyFirstInterface". Its properties should appear in the Inspector view.
6. In the Inspector view, click the Expand button.
7. Locate the `profile` property and click the button next to it to get the pop-up menu.

8. Scroll through the menu until you locate the GuiContentProfile and select that.
9. Click Apply.

   Now you have a Content Control to which you can add other controls.
10. Click the New Control button and choose GuiButtonCtrl from the pop-up menu.
11. Select the button using one of the two techniques you've learned (via the Content Editor or via the Control Tree).
12. Look in the Control Inspector view and locate the text property for this new control. Put some text in it.
13. Enter "quit();" in the command property.
14. Click Apply.
15. Click the Save button.
16. The Save feature will automatically use the top-level control in your interface for the file name, so leave that as is.
17. At the top of the Save dialog box is a button that you can use to select which folder in which to save the file. Choose the fps folder.
18. Click Save.

There, you've created an interface using the Torque GUI Editor!

Now let's break it! No…I mean, let's test it!

1. Open the console using the Tilde key ("~").
2. Type in the following, pressing the Enter key when you're done:

```
exec("fps/MyFirstInterface.gui");
```

3. Now type in the following, again pressing the Enter key when you're done:

```
canvas.setContent("MyFirstInterface");
```

Your interface should pop up on the screen. Just go ahead—press that button! Now you see that the whole program quits, because that's what you programmed it to do.

Of course, this is a simple interface. They can get as complex as you need. You can see that there is a lot of power available in Torque to address your interface needs. And if Torque doesn't have it, you can create it yourself!

Moving Right Along

So now you should have a reasonable understanding of how controls are made and added to an interface. You've seen the innards of some of the more common controls that are available in Torque.

You've also learned how to use one of the valuable built-in tools that Torque supplies, the GUI Editor. It's worth your while to practice making a few interfaces, even goofy ones that have no purpose if you like, just to reinforce the steps involved and to become comfortable using the GUI Editor.

Staying with the visual aspects of a game, in the next chapter we will examine structural material textures.

Team LRN

CHAPTER 11

Structural Material Textures

In earlier chapters we encountered textures used to enhance the 3D game environment in the resources included with the Emaga sample game. We only caressed the topic with the most feathery of touches. As the book progresses we'll explore the topic in depth from many different angles. In this chapter we'll look at one aspect of 3D game textures—those used to define 3D structures, like buildings, walls, sidewalks, and other virtual world artifacts.

You can judiciously and creatively use textures in several important ways. We'll use a prebuilt scene with a few basic and more complex structures to illustrate some of these principles, including the following:

- **Project information.** One of the most basic uses of textures in a 3D game is to define the object containing the textures. A simple box shape can become an electrical transformer, a house, a crate of weapons, or an air conditioner, merely by applying different textures to the shape.
- **Convey mood.** We can set a mood in a scene using different styles of textures. The amount of subtlety is up to the designers; a somewhat unremarkable and neutral air vent high on a wall can become an ominous clue to an unseen threat by adding a graphic of slime or other unmentionable stuff oozing from its louvers.
- **Establish space and place.** A cramped machine room full of noise and whirling parts might have shapes built with textures jammed with pipes, wires, knobs, and other mechanical items. The machinery shapes would probably be busy-looking affairs, even in static form. On the other hand textures for the walls in a high-ceilinged, multistory hall might have only vertically oriented lines and long, thin curves, with high-contrast shading.

Team LRN

During this chapter you will be directed to use Paint Shop Pro from time to time, so it's a good idea to have it open and ready for use.

Sources

There are many ways to create textures for use in structures. Techniques can range from the obvious (photographing buildings and walls and other real-world items or drawing them with pen and pencil) to the more imaginative (rubbings with paper and charcoal) to the more high-tech (using texture-creation software).

In this section we'll look at the two of the most accessible texture-creation methods, photography and original artwork.

Photography

To use photography as a source, you'll need a camera, of course. Digital cameras with decent resolutions available can be quite inexpensive. Most digital cameras come with hardware that allows you to quickly upload the images to your computer.

Digital versus Film

If you buy a digital camera, you should get one that will provide pictures of at least 800 by 600 pixels with 32-bit color.

Your other options are to use a normal film camera and then either scan the resulting photos or send the film to a shop that will digitize the photos for you when they're developed. These shops are quite common, and the extra step of digitization of your developed film is often a no-cost "loss-leader" that the shops use to attract business.

Scanners are also low-cost items. The minimum specification for a scanner that you need for use in game development would be a 600-dpi 32-bit color scanner. Flatbed scanners are best for this kind of work.

note

If you intend to use photography as a source for textures, be aware that there are some things to watch out for. Don't use pictures of items with trademarked images or copyrighted text or graphics on them. You will probably end up in violation of trademark or copyright law if your game ends up shipping with those images in it.

If your game absolutely must include a photo of a billboard ad for a popular soft drink, make sure you contact the soft drink company to obtain written permission before you ship your game. In addition to staying legal, you also might just be able to obtain sponsorship or some other support from the company for your game. Now I'll admit that this is probably not likely, but it is certainly possible.

When you have identified a candidate texture for use in your game, make sure to take several different pictures of the item from different angles, at different distances, and in different lighting, if possible. Take lots and lots of pictures, and then review them when you get back to your home or office to find what suits your needs. Keep all the originals. Sometimes, on later examination, you will discover details that you didn't notice when first taking the pictures. These details may require choosing a different shot from a different angle to ensure that they don't show. It wouldn't work to have the condensation trail of an airliner in an image used for the sky in a game that takes place in a medieval era.

Figure 11.1 shows a picture of some interlocking bricks used for a walkway. Figure 11.2 shows the same walkway with the picture taken under slightly different circumstances (for example, the area photographed for Figure 11.2 was a few feet away from the area shown in Figure 11.1). One detail I picked up quickly when examining the photos on my computer that I didn't notice at the scene is that some of the bricks in Figure 11.2 are double bricks, lain side by side. This despite the fact that I have trodden this particular walkway literally thousands of times in the past 10 years or so!

So the lesson is this: When at the scene, don't be a censor and don't be judgmental. Just take oodles of photographs of the items in question. Sort it all out back at the shop.

Figure 11.1 Real-world candidate for sidewalk texture.

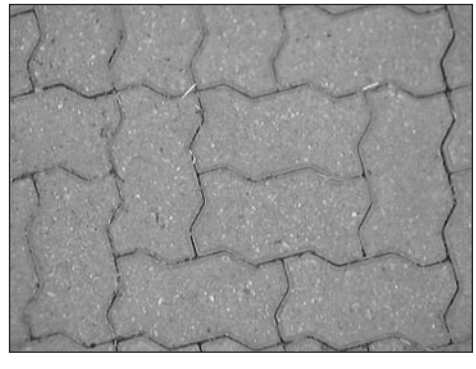


Figure 11.2 Alternate candidate for sidewalk texture.

Postprocessing

After getting back to the shop, you will probably have to do a certain amount of postprocessing of the chosen photos. Even if you were creating a photorealistic game, you would still need to ensure that the lighting and palettes of the textures were close enough matches to each other. This would be especially true when your texture photos were taken in different areas at different times.

It's probably best to do all of your pixel-related processing first, before you crop or extract your textures. This ensures that the changes you make are done in the proper context—the areas that you *aren't* interested in will change along with the areas you *are* interested in—thus guiding your efforts more appropriately.

All of the photo-processing capabilities of any tools you have are at your disposal. Three

Team LRN

specific operations are generally used more than the others: color matching, lighting, and cropping.

Color Matching

The first thing you will probably need to do is match the colors of your texture to existing in-game textures and lighting conditions. Usually you will match your colors by adjusting the illuminant temperature of the colors. This is easily done in Paint Shop Pro by choosing Adjust, Color Balance, Gray World Color Balance. The dialog box that appears does a good job of guiding you in your adjustments, showing you how to adjust to sunlight, fluorescent light, and so on.

Unfortunately, it is extremely difficult to illustrate the differences between different temperature settings in grayscale images like the ones used in this book, so Figure 11.3 may not adequately demonstrate the subtle variations. If you installed the companion CD, you can find the full-color version of the image in the file C:\3DGPAi1\RESOURCES\Ch11\11-03.tif. Of course, you can go ahead and try the various settings in Paint Shop Pro and see the differences for yourself.

In Figure 11.3, from left to right, the three settings chosen are Incandescent, Fluorescent, and Bright Sun. Compare these variations with the original shown in Figure 11.2 and found at C:\3DGPAi1\RESOURCES\Ch11\11-02.tif.

The light of bright sunlight on a clear day contains all colors of the visible spectrum pretty well in their natural proportions, with the sole modification being some filtering by the atmosphere. This atmospheric filtering (predominantly by water molecules) scatters a certain proportion of light at the blue-violet end of the spectrum, reducing the amount of light at those wavelengths that makes it to the surface. Nonetheless, you can see that there is still a strong blue component in the bright sunlight area.

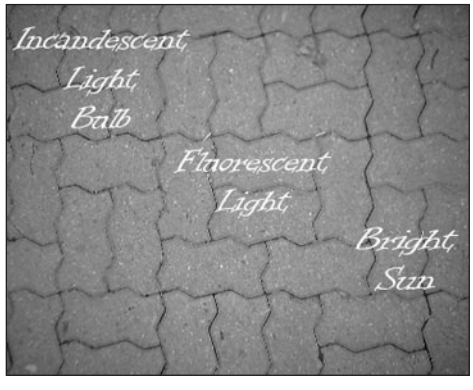


Figure 11.3 Reference image for three color temperatures.

The fluorescent light area shows a somewhat more balanced spectrum, with less blue than with the bright light. The incandescent light bulb area shows the opposite end of the spectrum from sunlight and has a much warmer feel.

So in Figure 11.3 the color temperature moves from warm on the left to cool on the right.

The original image, as shown in Figure 11.2, has a coloring somewhere between fluorescent and sunlight, leaning heavily toward the fluorescent. This is not really a surprise—I took those photos outside on a sunny clear summer day, but in

the shade. The illumination is therefore provided by the light reflected from the surroundings, which in this case has had the effect of moving the spectrum toward the middle.

Lighting

Lighting is closely tied to color matching. Changes in the apparent lighting of an image will tend to drag the color temperature in one direction or the other. So keep this in mind when you apply lighting changes to images.

In the context of processing 2D images for use as textures, what we are trying to achieve is imparting a sense of the light direction and light "play" upon the surface of the texture being portrayed.

For example, one feature about surface textures we may need to enhance is the sense of depth. As shown in Figure 11.4, the texture may contain numerous small stones that protrude from a flat surface.

One simple method we can use to increase the sense of depth is to increase the contrast. The problem with adjusting the contrast is that it tends to drastically alter the color temperature—the more contrast, the warmer the overall color temperature.

The obvious way to handle this would be to boost the contrast and then tweak the color balance. Sometimes this does not work so well, especially if a wide spectrum of colors is represented in the image. In those cases there are other ways to deal with the issue, such as tweaking the saturation.

What you see in Figure 11.4 is the original texture on the left and the adjusted texture on the right. In this case what I did was use Paint Shop Pro to enhance the contrast by 40 percent (choose Adjust, Brightness and Contrast, Brightness/Contrast) and then reduce the saturation by 41 percent (choose Adjust, Hue and Saturation, Hue/Saturation/Lightness).

Cropping

When using photographs as image sources, we rarely want to keep the entire image. Artifacts such as lighting changes at the periphery, fish-eye distortion at the edges caused by extreme perspective, extraneous items in the image, and other issues typically make the outer edges of photographs unsuitable for use as textures.

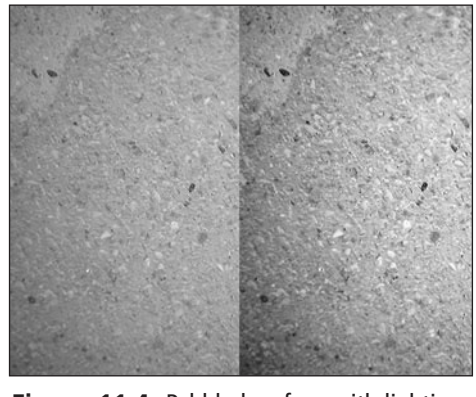


Figure 11.4 Pebbled surface with lighting adjustment.

The solution is to crop the image, leaving behind the portion that is useful to us. Figure 11.5 shows a piece of wood that has a texture of interest. It stretches across the entire frame from left to right but only covers somewhat less

than half of the vertical area. It's also not parallel with the sides of the image. In this case we will want to crop the wood out and "de-rotate" it as well.

You might be tempted to crop the wood out and then apply rotation to straighten it out, but experience shows that these operations should be done the other way around. Just as with the color and lighting operations, apply the geometric changes first, and then crop the texture. This allows the image processing software to make its geometry in the full context of the image parts that surround the area of interest, which can have a subtle effect on the end result.

Another reason for resolving the geometric appearance of the texture before cropping is that cropping tools tend to use rectangular shapes for selection. It is helpful to the overall process and productivity to crop images where the areas of interest are appropriately oriented horizontally and vertically.

Figure 11.5 A photograph that needs to be cropped.

Figure 11.6 Crop tool icon.

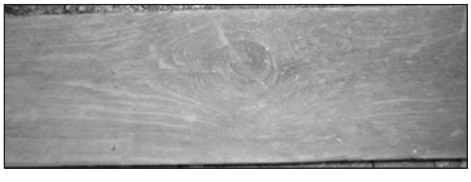


Figure 11.7 Cropped portion of unaltered photo.

To use the crop tool in Paint Shop Pro, click the Crop tool icon on the Tool palette (see Figure 11.6). Click and drag the tool on the image to select the rectangular area of interest. The resulting selection rectangle will have small square handles on the sides, which you can click and drag to resize the crop area. There is also a round handle in the geometric center of the crop area—click and drag it to move the entire crop area. When you are satisfied with your selected area, double-click the image to cause the actual cropping operation to take place.

Figure 11.7 shows the result of merely cropping the image to include all portions of the wood without first altering the orientation of the wood. It still needs to be rotated. Of course, you may actually want the woodgrain to be slanted, but then you may need to remove the nonwood slivers of area above and below the woodgrain by erasing them to a fixed solid color, or making those areas completely transparent.

In our case we really want the woodgrain to be parallel to the bottom and top edges of the image, so we should rotate the woodgrain portion before cropping. Use the rectangular Selection tool (see Figure 11.8) to select the area to be rotated.

Then choose Image, Rotate, Free Rotate to get the Free Rotate dialog box (see Figure 11.9).

Click the Right button in the Direction frame, and click the Free button in the Degrees frame. Finally, type **1.00** in the text box next to the Free button, and click OK. This will rotate the selected area 1 full degree to the right (see Figure 11.10).

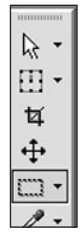


Figure 11.8 Rectangular Selection tool icon.

You should have your rotated area with the selection marquee still surrounding it. Don't touch anything yet—leave the selection as it is.

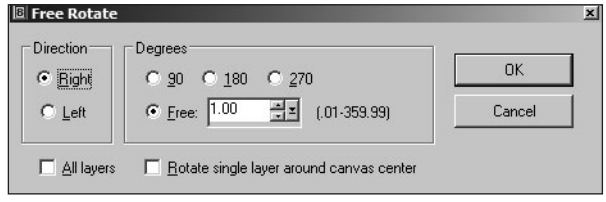


Figure 11.9 The Free Rotate dialog box.

Now after having explained the Crop tool, I'll show you another way to crop the image that is sometimes more convenient than using the Crop tool. With the rotated area still selected, choose Image, Crop to Selection and the image will be cropped for you! You will then end up with an image as shown in Figure 11.11 suitable for use as a texture.

Now compare Figure 11.11 with Figure 11.7 and you will see the difference.

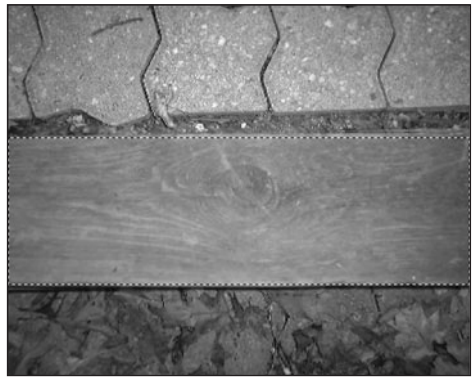


Figure 11.10 The rotated woodgrain.

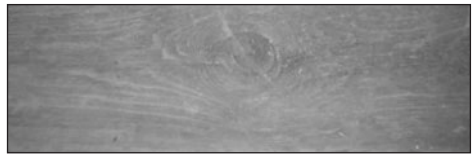


Figure 11.11 The cropped woodgrain image.

Original Artwork

The other approach to creating textures is to use original artwork. Some people believe this is not a real option for them, because they think they can't draw or paint to save their lives. I tend to feel that everyone can learn the techniques required. My intent here, however, is not to teach you how to draw, so if you want to learn more, I encourage you to look into taking some lessons.

If you are satisfied with your artistic skills, then you have another rich avenue for texture generation available to you. The techniques used to convert a photograph to a texture can also be used to convert your hand-made images to textures.

Another approach for creating original artwork is to create your images directly in a tool like Paint Shop Pro. You can draw freehand using the mouse or a pen tablet.

With tools like Paint Shop Pro you have a wide variety of means for creating textures, including a specific Texture Effects tool in the Effects menu, as shown in Chapter 8. Figure 11.12 shows examples of textures created using the built-in features of Paint Shop Pro. I encourage you to explore this tool in depth. It can really be a timesaver. And you can use it to create some knockout textures.

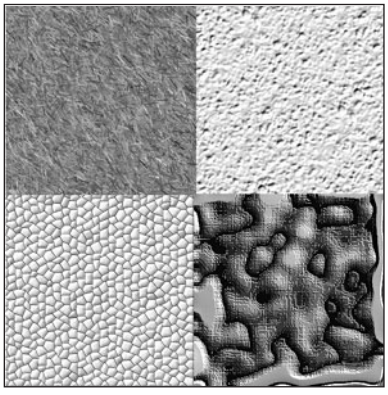


Figure 11.12 Example textures.

Scaling Issues

When creating your textures, you will need to pay attention to the issue of scale. The sizes of the things within an image that is used to make a texture have a particular relationship to other real-world objects. We are subconsciously aware of many of these relationships from our exposure to the world in general and will notice when the textures are out of proportion to the items they adorn. If it's bad enough the effect can sometimes be similar to the sound of fingernails being dragged across a chalkboard!

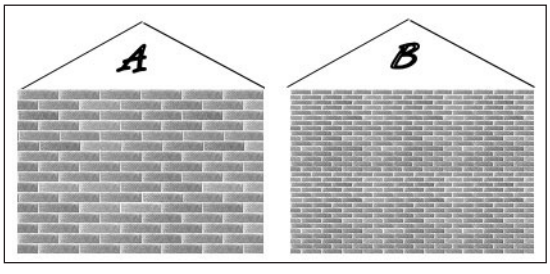


Figure 11.13 Scaling bricks.

Figure 11.13 shows two stylized houses. The bricks in house A are far too large, while the bricks in house B are more appropriately sized, yet may still be a bit too large. Yes, there are some uses for stone blocks having proportions such as those in house A, but they are rarely used in bungalow-sized or two-story homes, as depicted in the figure.

Figure 11.14 Scaling error.

The scale issue can pop up anywhere, as you can see in Figure 11.14. The texture image in the corrugated metal bridge surface is probably about 10 times larger than is appropriate. Sometimes you might need to redo the texture to match—other times you can adjust how the texture is applied to the polygons using the modeling tools. My rule of thumb is that if the texture image size is 64 pixels by 64 pixels or smaller and needs to be made larger, you should make a new texture at the larger size. The same goes the other way: If the image size

Team LRN

is larger than 64 pixels by 64 pixels and needs to be made smaller, then make a new texture at the smaller size.

Tiling

Many structures have large surfaces with repeating patterns. The best way to approach making textures for these surfaces is to create one smaller texture that is replicated many times across the surface, rather than simply making one large texture.

The replication will usually take place in two dimensions. It is important to make sure that the edges of the texture align properly when they meet. Figure 11.15 shows this to good effect. You can see the obvious horizontal as well as the more subtle artifacts in house A where the tiled brick textures don't quite line up. In house B, where care was taken to ensure that the texture edges matched up correctly, those artifacts aren't visible.

However, in house B in Figure 11.15 there is another obvious artifact of tiling, this time caused by asymmetric lighting effects in the texture shading. You can see each repeated texture tile—its position is marked by the presence of the darker shaded bricks in a repeated pattern. This effect can be quite subtle and difficult to detect in an image viewed in isolation.

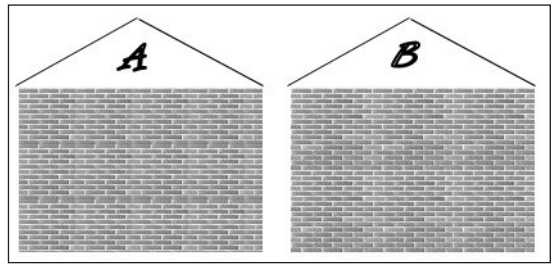


Figure 11.15 Tiled brick texture.

Figure 11.16 shows the texture used in house B of Figure 11.15. Looking at it in isolation, you would be hard pressed to notice the subtly darker shaded bricks.

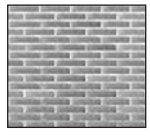


Figure 11.16 The brick texture with asymmetric shading.

The simplest way to fix up a texture for use as a tiled texture is to copy the left edge, about 5 or 10 pixels wide, mirror the copy horizontally, and then paste the copy on the right side of the image. Do the same for the bottom edge. Of course, you can go from top to bottom or right to left as well. The important step is the mirroring.

After placing the mirrored edges, spend a little time blending their inner edges with the interior portions of the image.

Figure 11.17 shows a stone block texture that is a candidate for use in a tiling situation.

Figure 11.17 A stone texture.

Figure 11.18 shows the texture tiled in a set of four. Again, you can see the artifacts caused by the mismatched edges.

Figure 11.18 Poorly tiled stone texture.

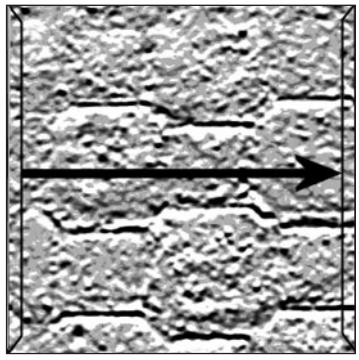


Figure 11.19 Replicating the left edge.

Figure 11.19 shows the left edge being copied, mirrored, and placed on the right.

Figure 11.20 shows the same thing happening with the bottom edge.

Finally, Figure 11.21 shows the tiled result.

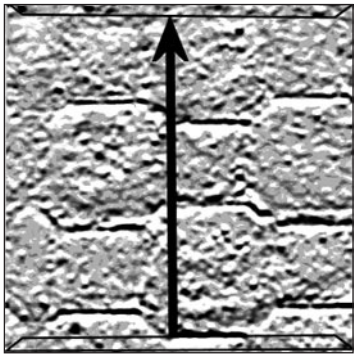


Figure 11.20 Replicating the bottom edge.

Figure 11.21 Properly tiled stone texture.

Texture Types

There are far too many texture types and classes of material appearances for me to enumerate them with any sort of thoroughness. Given that, there is a much smaller set of texture types that are found over and over in nature and man-made structures.

Most of the following textures are types that are used for buildings, bridges, and other man-made items in a game world.

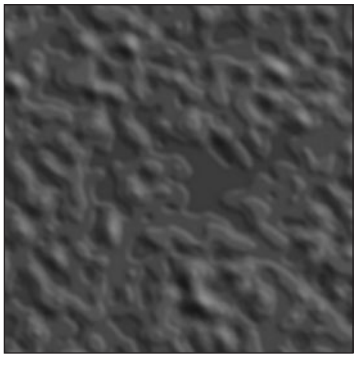


Figure 11.22 An irregular texture.

Irregular

Irregular textures tend to have a general disorder and random appearance, like that shown in Figure 11.22. Dirt and grass are examples of irregular textures. Quite often irregular textures are combined with other, different irregular textures in order to give a weathered or damaged appearance to an area or surface.

Rough

Rough textures, as shown in Figure 11.23, sometimes have somewhat the same sense about them as irregular textures. They are often used as tiles on a surface like a sidewalk or rough concrete walls.

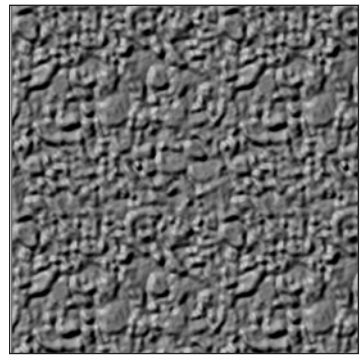


Figure 11.23 A rough texture.

Pebbled

Pebbled textures are another example of textures often used for paved surfaces and stone walls. Tarmacadam pavement is an example of a finely pebbled surface when viewed from a distance of about 5 or 6 feet. Figure 11.24 shows a more obvious pebbled texture that could be used for a wall or decorative planter.

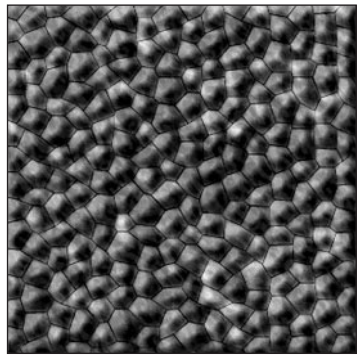


Figure 11.24 A pebbled texture.

Woodgrain

Figure 11.25 shows a woodgrain texture that has many highly variant bundles of lines ranging from fine to coarse that run roughly parallel to each other, sometimes interrupted by swirls and knots. Some kinds of stone have similar appearances.

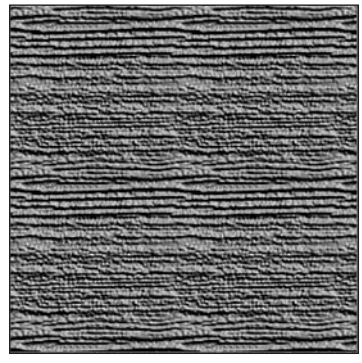


Figure 11.25 A woodgrain texture.

Smooth

We all know when something is smooth—there are no discernable bumps or irregularities to our touch. Depicting smoothness in textures can be a little difficult. We usually create a rather bland surface look and then introduce a few soft and mild irregularities in order to emphasize the smoothness. Figure 11.26 shows a smooth texture.

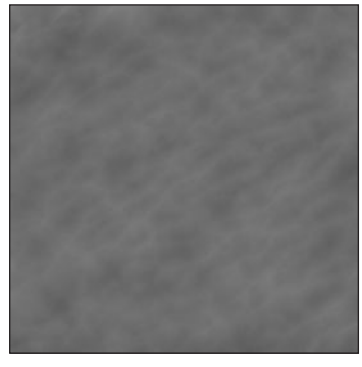


Figure 11.26 A smooth texture.

Patterned

Patterned textures are pretty straightforward. The intent is not necessarily to convey the contour, bumpiness, or feel of a surface, but rather to represent regular shapes or patterns that appear on an item. Figure 11.27 depicts a pattern that could be used to represent the louvers of an air duct in a wall.

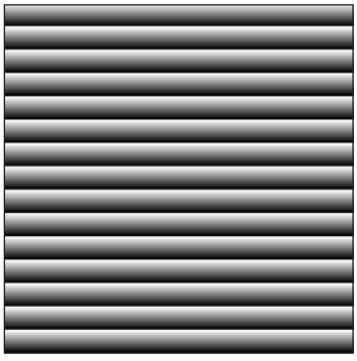


Figure 11.27 A patterned texture.

Fabric

Fabric textures emulate the appearance of things like canvas or carpet. Fabrics may be woven or not, but they all tend to exhibit fine repetitive shapes. Figure 11.28 shows a woven fabric texture that could be canvas.

Figure 11.28 A fabric texture.

Metallic

Metallic textures tend to have a dominant color, with a strong dark shadow that follows the outer contours of the metallic object and a bright accent color that runs along raised surfaces. Figure 11.29 shows a texture that could be used for a metal tube.

Reflective

A reflective texture simulates the effect of a light source in the scene reflecting strongly off the surface of the textured object. Figure 11.30 is such a texture that might be depicting a bright overhead light reflecting off a window.

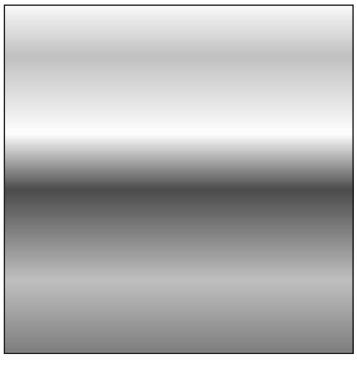


Figure 11.29 A metallic texture.

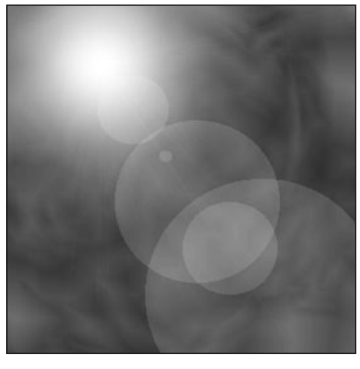


Figure 11.30 A reflective texture.

Plastic

Plastic textures are similar to metallic textures in their manner of shading and highlighting. Plastic tends to have more of an oily appearance to it at times, so the shading and highlights are often more sinuous. As shown in Figure 11.31,

the highlights tend to be less clearly defined than with metallic textures, while the light source often appears as a distinct highlight.

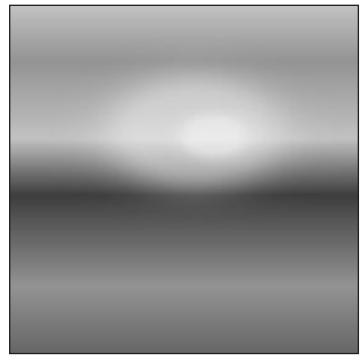


Figure 11.31 A plastic texture.

Moving Right Along

In this chapter, we examined how to collect images to use in applying textures to objects that represent real-world structures. We saw some of the processing techniques that we may need to use to prepare our images for use as textures, like color matching and cropping.

Some of the areas that can be more problematic when considering textures for structures are scaling the images and preparing them to be "tiled" if the texture will be used in a repeating fashion. A texture that can be tiled is one whose opposite edges can be mated together without producing a noticeable seam.

Finally, we explored some of the more common texture patterns and characteristics that are used in games.

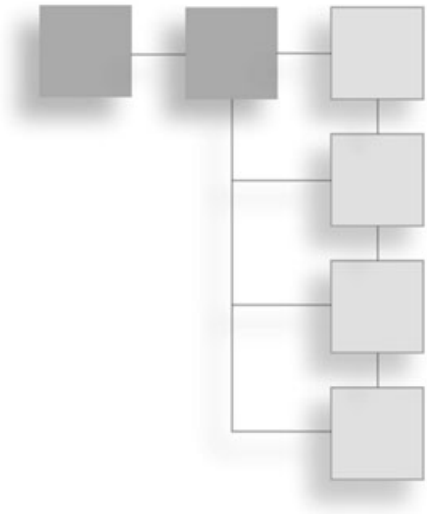
In the next chapter, we will look at terrains, which are often used to provide that touch of realism in our game worlds. Some of the ideas we've covered in this chapter will certainly be useful in the next chapter as well.

This page intentionally left blank

Team LRN

CHAPTER 12

TERRAINS



Many games take place exclusively inside buildings or structures, like tunnels. And many other games involve exclusive outdoor game play. Then there are some games that have a mix of each.

When your game has an outdoor component, you need to represent the *terrain*, which in game terms is the combination of the topography (hilliness, for example) and ground cover (grass, gravel, sand, and so on). The topography is modeled using a 3D model, and the ground cover is represented by textures.

In addition to representing the ground, you also need to represent the sky, if you want to have interesting outdoor game play. Typically, a construct called a *skybox* is used to represent all of the sky, from horizon to horizon.

Terrains Explained

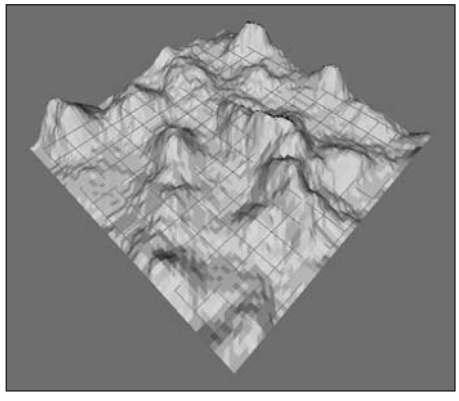


Figure 12.1 An untextured terrain tile.

To understand terrains in a game development context, we need to look at the characteristics of the terrain we want to model. These characteristics will drive our need for the data that defines the terrain we want to make and therefore will heavily influence how and where we obtain that data.

Terrain Characteristics

A basic unit of terrain is the *tile*. Essentially a terrain tile is a collection of polygons that form a 3D model that represents the terrain, as depicted in Figure 12.1.

Team LRN

When we model terrain in a game, there are a number of choices we have to make. We need to decide the level of terrain *fidelity* we want to achieve. Another choice is to figure out the *spread* of the terrain. Finally, we need to decide what sort of *freedom* the terrain embodies. Table 12.1 lists characteristics and the ramifications of each of these choices.

Table 12.1 Terrain Characteristics

Characteristic	Description
Fidelity	*Terrain fidelity* measures how accurately the terrain reflects real topography found somewhere in the world—how realistic it is. The realism can be reflected in both the modeling and the textures. Modeling fidelity can be described as any of the following: **Realistic**: Accurate at 1:1 scale in all dimensions with high-resolution textures representing the terrain cover. **Semirealistic:** Accurately scaled, usually to a smaller size. Often the vertical scale is 1:1 while the horizontal scales are around 1:2. The game *World War II Online* by Cornered Rat Software has all of Western Europe modeled in this fashion. The game uses medium-to-low resolution textures to represent ground cover. **Quasi-Realistic:** Not accurately scaled in any dimension, but still attempts to represent a real location in the world. Usually employs high-resolution ground cover textures. The scales and textures are chosen to give a sense of the locale that works well in the game environment. NovaLogic's *Delta Force* series takes this approach. **Unrealistic:** Everything else! Unrealistic terrain is most commonly used to specifically enhance game play or the backstory of the game.
Spread	*Terrain spread* is the degree to which areas of the terrain are unique. Terrain is created in units called tiles. The spread is related to these tiles in one of three ways: **Infinite:** A square terrain region is repeated, or tiled, in all cardinal directions, such that when the player leaves a region to the west, he enters a new copy of the same terrain tile from the east. This continues for as long as the player keeps moving in that one direction. **Finite:** The terrain tiles are repeated in all directions, but at some point the repetition stops. **Untiled:** Terrain tiles are not repeated.
Freedom	*Terrain freedom* is the measure of how much the player's in-game movements are restricted by the terrain. Terrain freedom is closely coupled with terrain spread. There are really only two degrees of terrain freedom: **Closed:** Closed terrain limits player movements in all cardinal directions at some point. With closed terrain, at some point after a player has been moving in a particular direction, he cannot continue that way, either because there is a virtual physical barrier or because the program prevents further movement. In any case, the terrain is usually modeled beyond the barrier only as far as the player can see. After that—nothing. **Open:** Open terrain allows player movement in any direction for as long as the player wants. Some games will warp the player to the "other side" of the world, where he will keep crossing terrain tile copies until he returns to the place he started.

Team LRN

There are practical considerations that direct our terrain design choices. Many game engines simply aren't capable of handling the distances involved in large-scale terrains or the number of objects required to appropriately populate them. Some game genres aren't suited to open terrains—the player needs to be confined in order to advance the game story as required.

Terrain Data

When you want to create a high-fidelity terrain model of a real place in the world, you are going to need to get the data from somewhere. If the area in question is small enough, you may be able to go out and gather the information yourself if you're handy with a theodolite (a surveyor's tool). You might be able to glean the necessary information from topographic maps. In either case there is a lot of work involved in the data gathering phase alone. You will need accurate distance measurements and altitudes, as well as photos of the ground cover.

But don't despair! There are sources for high-resolution terrain information available on the Internet. If you go to http://edcwww.cr.usgs.gov, the Web portal for the United States Geological Survey (USGS; part of the U.S. government), you can find a wealth of terrain data.

The data is available in several forms, but the standard form is the DEM (*Digital Elevation Model*). DEM-formatted data files have the .dem file extension. Another format in use is the DTM (*Digital Terrain Model*), which uses the .dtm file extension. Finally, a powerful and complex format called SDTS (*Spatial Data Transfer Standard*) also exists but is not in wide use outside of scientific niches. SDTS files are denoted by the .ddf file extension.

In any event, the ground cover information is not included in these various model formats, so you'll need to gather that as well. Again, the USGS comes in handy with its satellite imagery—some of it taken down to a resolution of less than a meter per image pixel.

DEM files provide elevation information for specific coordinates of places on Earth. DEM files can be converted to a format used by game engines called a *height map*. We won't go into detail about how to use DEM data for your game, but you can use several of the resources listed in the appendixes to locate the data and tools needed.

Terrain Modeling

There are basically two approaches that 3D game engines use to model terrain in a 3D world. In both cases 3D polygon models represent terrains.

In the *external* method we include the terrain as just another object in the game world. This method offers much freedom of manipulation. You can rotate the terrain model, skew it, and otherwise subject it to all manner of indignities. All 3D engines support this approach. While flexible, it is usually an inefficient way to render complex large terrains.

Team LRN

The second approach is the *internal* method, where terrain is rendered by special code in the game engine often called a *Terrain Manager*. Using the Terrain Manager approach allows game engine programmers to apply specific memory and performance optimizations to the terrain object, because they can discard unnecessary functions that would be available to general-purpose objects. Because of this, Terrain Manager terrains can sometimes be made larger and more complex than those created using other approaches.

Most 3D engines, like Torque, that use a Terrain Manager also provide terrain generation, manipulation, and editing tools that we can use to create our own terrains. Usually importing height maps is available for terrain generation. Some engines, like Torque, have built-in Terrain Editors that allow the game developer to directly manipulate terrain polygons, within constraints, to create the desired hills, valleys, mountains, and canyons.

Height Maps

Figure 12.2 depicts a height map. As you can see, it's a grayscale image. The 2D coordinates of the height-map image map directly to surface coordinates in the game world. The brightness off each of the pixels in the image represents the altitude at that pixel's location—the brighter the pixel, the higher the elevation. Usually we use an 8-bit per pixel format, which means that 256 discrete elevations can be represented.

The concept is an elegant one and not difficult to grasp. If you are familiar with viewing topographic charts and maps, you'll find that height maps have a familiar flavor to them, even though the contour lines are missing. One of the deficiencies of height maps is the resolution (as you can see in Figure 12.2). To represent a geographic locale that is 1 kilometer square, a height map that represents 1 square meter as a pixel needs 1,000 pixels per side, for a total of 1 million pixels—big, but not too large. If I want to increase the terrain area to cover 16 square kilometers (4 meters per side), then I need to store 16 million pixels. At 8 bits per pixel, that equals about 16MB of data. If we want to model the terrain for an area that is 10 kilometers per side, we are looking at 100MB of storage!

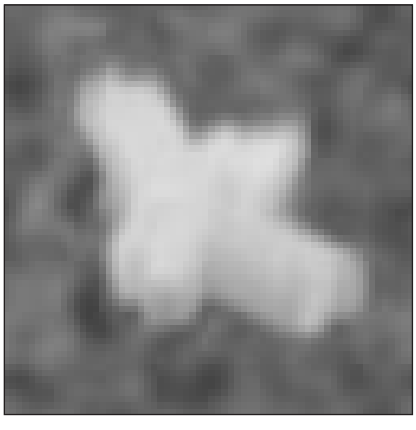


Figure 12.2 A terrain height map.

We can, of course, reduce the terrain resolution—let's say, have a pixel equal 4 square meters in the game world. This would chop those 100MB back to 6.25MB. However, that gain is offset by the fact that our terrain will now be blockier and less realistic.

Figure 12.3 shows a terrain model generated from the height map shown in Figure 12.2. In this case MilkShape was used to import the height map and create the terrain object.

Terrain Cover

In the simplest sense, *terrain cover* refers to all the stuff that you find on the ground, including:

- grass
- flowers
- dirt
- pebbles
- rocks
- trash
- litter
- pavement
- concrete
- moss
- sand
- stone

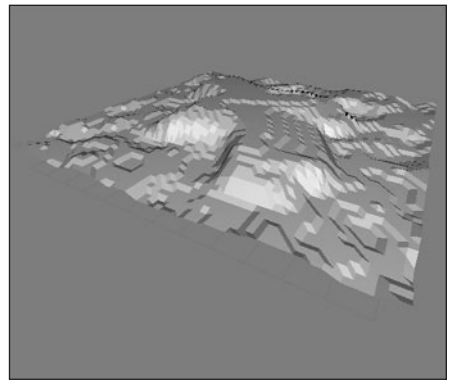


Figure 12.3 A terrain created from a height map.

Obviously this is not a comprehensive list, but it does demonstrate the point.

We represent the terrain cover with textures. Our options for creating these textures are much like those we considered when we created textures for structures in Chapter 11—and the factors that dictate which way to choose are also similar. It boils down to the terrain characteristics in the game that matter to you.

We can also mix terrain cover textures in adjacent areas to portray a particular locale. It's a good idea to develop your own library of generic terrain cover for use in various situations.

Figure 12.4 illustrates some of the possible varieties of terrain cover. From left to right in the top row you can see grass, sand, and an intermixed sand and grass texture. In the bottom row from left to right are dirt, a muddy track, and eroded wet sand.

Tiling

Unless you are going to create specific terrain cover textures for every square inch of terrain, you will end up tiling your terrain cover at some point. All the issues brought up with tiling in other contexts apply here, such as matching texture edges to get seamless transitions and ensuring lighting in the textures is both appropriate and uniform. Additionally,

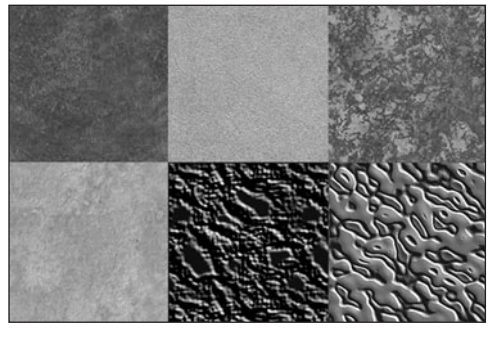


Figure 12.4 Some example terrain textures.

you should ensure that there are no patterns or marks in the texture that will stand out too much when the texture is repeated.

In Figure 12.5 you can see a repeating light pattern that tends to overpower the otherwise pleasing and pastoral scene. (Okay, okay, it *would* be pastoral if a storm wasn't brewing beyond the, um…Mountains of Evil in the distance. But besides that…)

The culprit in this case is the grass texture used, which is shown in Figure 12.6.

Notice the area of lighter grass, which is quite noticeably different from the rest of the image. When repeated over and over across large swaths of terrain, that feature detracts from the intended overall effect. We can enhance the image to minimize the problem, perhaps with something like that shown in Figure 12.7.

The result is dramatic and the difference is quite obvious, as you can see in Figure 12.8. Now, I confess that the texture could be better, but you have to admit that it is light-years ahead of the first version, shown in Figures 12.5 and 12.6.

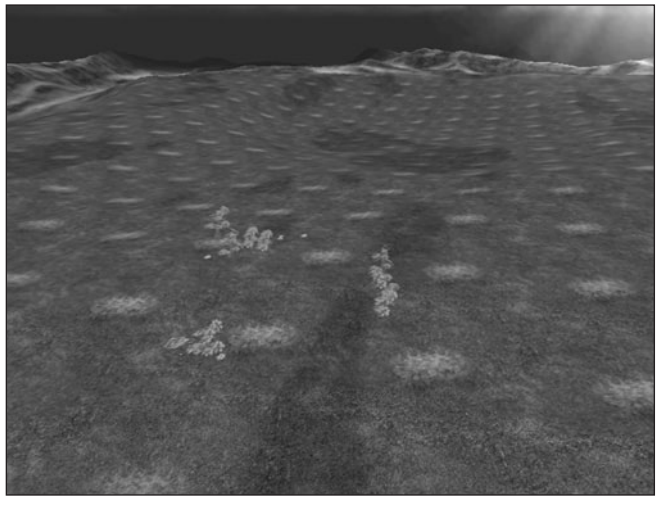


Figure 12.5 A terrain with tiling artifacts.

Creating Terrains

Okay, enough talk. Time for some action—let's create some terrain. We'll use the Torque Engine and its internal Terrain Manager to create the terrain, and we'll employ the height-map method using the in-game Terrain Editor. There is another method, direct manipulation, that we'll use later in Chapter 18.

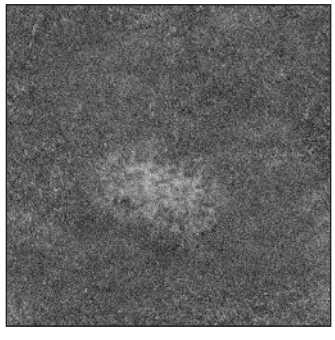


Figure 12.6 A texture with an undesirable feature.

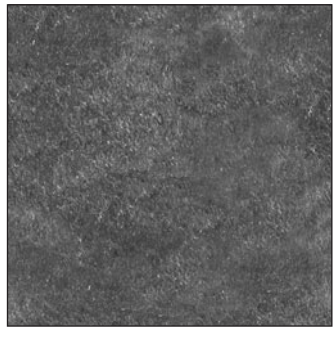


Figure 12.7 A texture without the undesirable feature.

The Height-Map Method

For this section, you will need to fire up Paint Shop Pro. You should be fairly familiar with the basics, so I won't hold your hand too much with respect to PSP operations.

note

The default size for a terrain in Torque (when the `squareSize` property in a MIS mission file is set to 8) is 65,536 world units (WU).

One WU in Torque is equal to one unit in most third-party map editors. A WU is equivalent to one scaled inch.

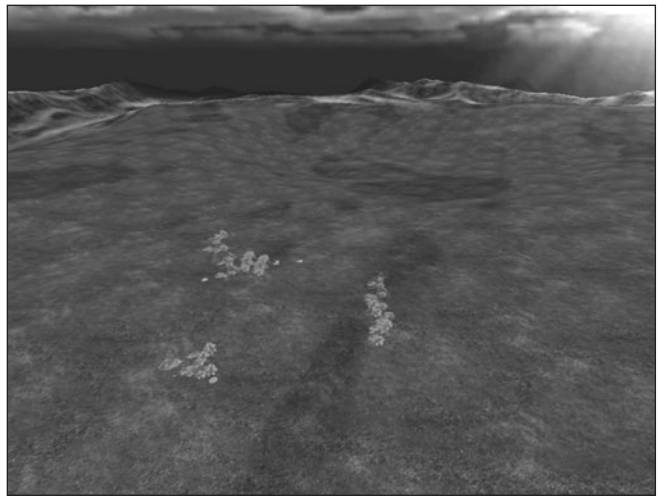


Figure 12.8 The terrain with improved tiled texture.

1. Start with a drawing of the contours to create the height-map image.

   If you have a source for colored contour drawings for a section of land drawn at full scale (1:1), such as shown in Figure 12.9, get one that suits your needs. If not, you can use the images shown here, but in their colored format, which you will find at C:\3DGPAi1\RESOURCES\CH12. Each image has the same name as the figure number used here.

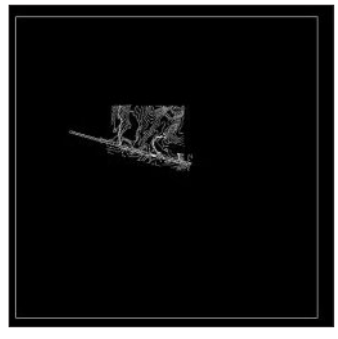


Figure 12.9 Contour map.

2. Clip out the portion you want and save it as a PNG image, as shown in Figure 12.10.

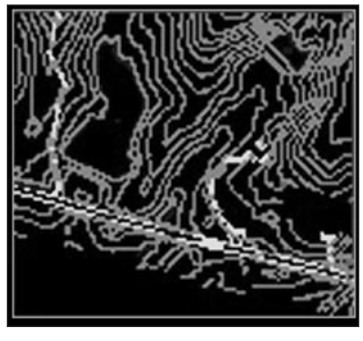


Figure 12.10 Cropped and resized contour map.

3. Now you need to do a little noodling over scale and unit numbers.

   In Torque each terrain square is made of two terrain triangles sized at 256 WU by 256 WU; as mentioned earlier, the default `squareSize` property in a Torque mission file equals 8 by default. The terrain has 256 of these squares per side for a total of 65,536 world units (inches) per side.

   256 WU×256 squares=65,536 WU (inches)

   If we convert the units, we get 5,641.3 feet, or 1,664.6 meters (1.034 miles, or 1.6646 kilometers).

   65,536 inches÷12 inches=5,461.33 feet
   1 mile=5,280 feet
   5,461.33 feet÷5,280 feet per mile=1.034 miles
   1 mile=1,609 meters
   1,664.6177 meters÷1,609 meters per mile=1.034 miles

The value 8 (for `squareSize`) and the value 65,536 (for terrain size) are not accidental; they are powers of 2. This works nicely with our images as well as the software. The size for our height-map image must be 256 pixels by 256 pixels. This means that when the image is stretched to fit our terrain of 65,536 inches by 65,536 inches, each texture pixel (*texel*) determines the horizontal distance of 256 inches (or 6.504 meters) of terrain. Because each terrain square is 256 WU, each height-map texel is used to determine the height of one terrain square.

256 pixels×256 WU (inches)=65,536 WU (inches)

256 inches÷39.37 inches per meter=6.5024 meters

6.5024 meters×256 pixels=1,664.6177 meters=1.664 kilometers=1.034 miles

4. Based on the preceding calculations, we can get the equivalent area in the image—crop the image just inside the line box created in the Figure 12.10 drawing representing 1.034 square miles.
5. Resize the image to 256 by 256 pixels.
6. Save the image as a PNG file to preserve the original colors for the contours.

   In a moment you will paint over this contour image using gray color values representing the heights of the contour lines. In this case the contours range from an elevation of 410 feet to 485 feet. This information is available from the source of the contour maps. The grayscale can be any sequence of gray RGB values within the 256 colors ranging from 0,0,0 (black) to 255,255,255 (white).
7. Establish your scale, keeping in mind that it's best to have some separation between the incremental values so they can be easily seen as you paint the contours.

   Examination reveals that there are 16 elevation increments in the contour range of 410 to 485. Divide the 256 colors for the grayscale range by 16, and you will get the values in Table 12.2, which starts at the color (0,0,0) and works up.

   Now that we have the values, we need to create what Paint Shop Pro calls *swatches*. We need to make a different swatch for each increment.
8. Make sure that you have the Materials palette visible in Paint Shop Pro by choosing View, Palettes, Materials and clicking on the Swatches tab in the Materials frame.
9. Delete any existing swatches by clicking on each one and then clicking on the trash can icon below the swatches. If you think you can keep track of your custom swatches and the ones already there, then you can skip this step.

Next we will create a new swatch for our first increment.

1. Click the Create New Swatch button at the bottom of the Swatch frame.
2. Type in a name for your swatch. I suggest you use either the increment number or the elevation value for the name. You could also use both, as in 1-485 for the topmost entry from Table 12.2.

Team LRN

Table 12.2 Elevation RGB Values

Elevation	RGB	Increment
485	240,240,240	1
480	224,224,224	2
475	208,208,208	3
470	192,192,192	4
465	176,176,176	5
460	160,160,160	6
455	144,144,144	7
450	128,128,128	8
445	112,112,112	9
440	96, 96, 96	10
435	80, 80, 80	11
430	64, 64, 64	12
425	48, 48, 48	13
420	32, 32, 32	14
415	16, 16, 16	15
410	0, 0, 0	16

3. The Color dialog box will appear. Here you type in your RGB values, referring to Table 12.2 for your numbers. Click OK to close the dialog box.
4. Repeat steps 1, 2, and 3 for each increment in the table.
5. Now fill in your image following the contour lines as shown in Figure 12.11. Use a combination of the Brush and Fill tools, at your discretion, to complete the task.

   Notice that in Figure 12.11 the grayscale value is the same at all the edges. This is because we want the edges to match when the terrain repeats itself, if it is tiled—and in this case that's what we will be dealing with. The edges could be different values; you would then just match them at the top and bottom or left and right sides.
6. When you have finished the "paint-by-number" process, convert the image to grayscale by choosing Image, Grayscale.
7. Save your image as a PNG file.
8. Flip the image around its X-axis—this flips the top with the bottom—by choosing Image, Flip. You should get an image like that in Figure 12.12. Make sure you save your work.

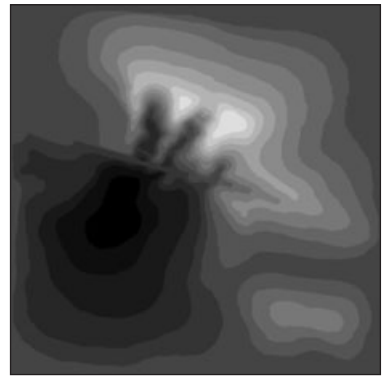


Figure 12.11 Contour map with grayscale values.

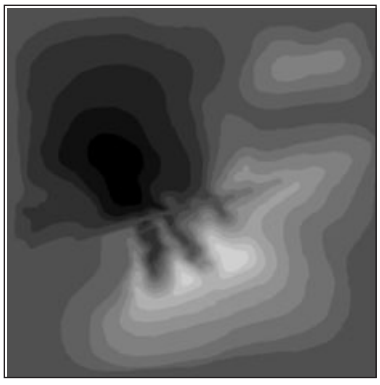


Figure 12.12 Terraced height map.

Notice the terrace effect in Figure 12.12. If you import this into Torque as is, you will have a set of terraced, or stepped, surfaces. If this is what you want, then you're good to go already. However, let's go a bit farther.

9. Make a copy of the image you created in step 7 to work with.
10. Choose Adjust, Blur, Gaussian Blur to smooth out the edges a bit. Use a radius of four and then save your changes to this new image as a PNG file. You should get an image much like the one shown in Figure 12.13.

tip

It can be more difficult to locate your original contour features after smoothing with Gaussian Blur. A quick work-around is to try reducing the radius or use the original image unblurred and smooth the terrain in Torque using the Terrain Editor (covered later).

A more time-consuming technique (but much more accurate and rewarding) is to create the terrain image at a much larger scale and reduce it to 256 by 256. For example, you might try constructing the image at around 2,048 by 2,048 or 4,096 by 4,096; this means *much* more painting time, but after reducing the image size again, the blending information is retained (although somewhat smoothed) by the resize algorithms. The resulting terrain is much more accurate than the Gaussian Blur process.

This last height-map image is the one you will work with to create the terrain.

Next, we will import these images into Torque.

11. Place the images in Torque's C:\3DGPAi1\common\editor\heightscripts folder. If the folder does not already exist, create it.

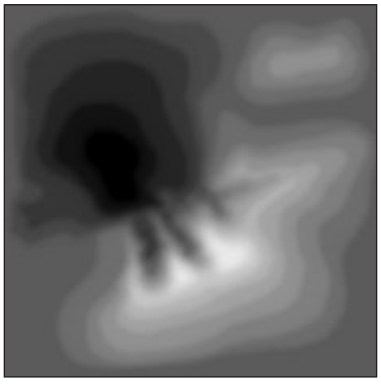


Figure 12.13 Blurred height map.

12. Use the Run fps Demo shortcut to launch Torque.
13. Run any existing mission to which you'd like to add this terrain or choose File, New Mission.
14. Press F11 to open the World and Terrain Editor.
15. Choose Window, Terrain Terraform Editor (as shown in Figure 12.14) to open the Terrain Terraform Editor.
16. On the right side of the screen, in the General Settings area (see Figure 12.15), set Min Terrain Height and Height Range in meters.

The maximum elevation in the terrain we are modeling is to be used for Minimum Terrain Height (the Minimum Terrain Height box is mislabeled in the Editor). You will recall that the highest elevation is 485 feet; this translates to a Minimum Terrain Height value of approximately 148 meters.

485 feet÷3.281 feet per meter=147.8208 (148) meters

Height Range represents the distance from our lowest to highest elevation. The grayscale color values of our height-map image will be interpolated between these values. We need to calculate the difference and multiply that by the ratio of highest color number divided by total number of grayscale colors (256) and convert to meters. Clear as mud?

485 feet−410 feet=75 feet

240÷256×75 feet=70.3 feet (240 is our highest color number in Table 12.2)

70.3 feet÷3.281 feet per meter=21.4 (21) meters

17. Now click the Operation box to roll out the Operation dialog box, as shown in Figure 12.16.
18. Select Bitmap from this dialog box—this brings up a bitmap Load File dialog box, as shown in Figure 12.17.
19. Highlight the image you want translated to a new terrain and click the Load button. You should find the height-map image you saved earlier in C:\3DGPAi1\common\editor\heightscripts, from Paint Shop Pro.
20. Click the Apply button at the right side of the menu. You will see the terrain change. To relight the scene, choose Edit, Relight Scene. There will be a slight pause in input response while the relighting occurs.
21. In the lower left of the screen the overhead map view of the terrain will change to show the contours imported from the height-map image.

    Notice that this image, as depicted in Figure 12.18, has the same orientation as the original one before we flipped it around the X-axis in step 8 of this list.

Figure 12.14 World Editor Window menu with Terrain Terraform Editor checked.

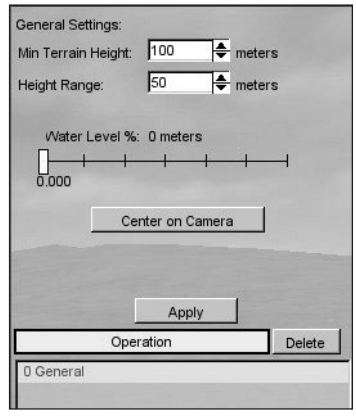


Figure 12.15 Terraform Editor.

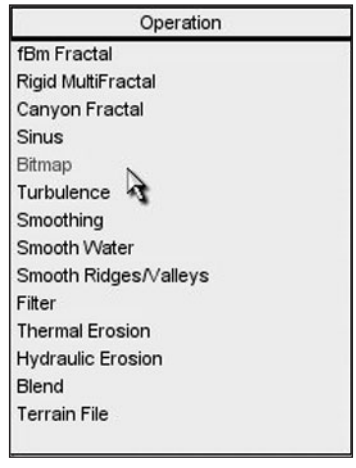


Figure 12.16 The Operation dialog box.

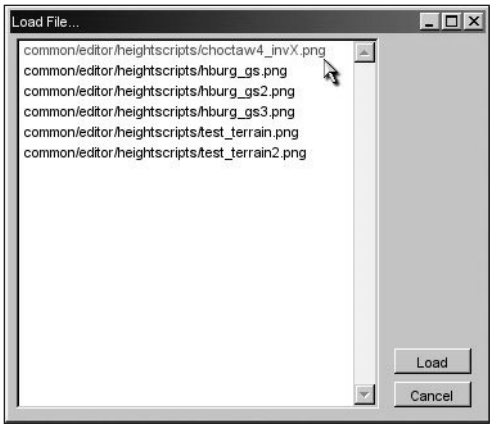


Figure 12.17 The Load File dialog box.

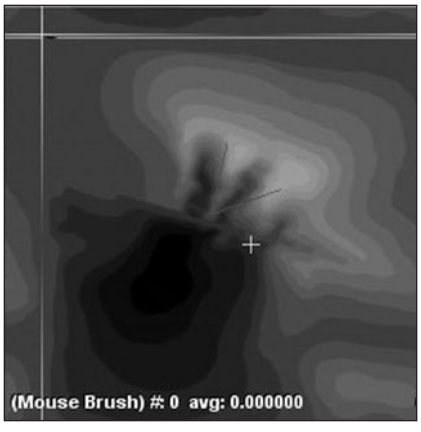


Figure 12.18 The overhead view.

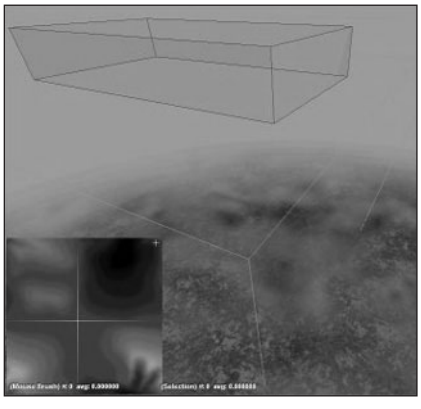


Figure 12.19 The terrain boundary.

The white line in the map shows the terrain boundary, representing the extents of your terrain before repeating. In the main 3D view, a green translucent box illustrates this boundary, as you can see in Figure 12.19. The terrain boundary is a fixed dimension—you can't change it.

Figure 12.20 illustrates where to find the inner red box that represents the mission area. You can change the extents of the mission area boundary by using the Mission Editor.

22. Choose File, Save As to save your mission with your own unique name. You should save your new file in the directory C:\3DGPAi1\fps\data\missions.

When you save your mission, the terrain data is also saved as a TER file in the C:\3DGPAi1\fps\data\missions directory. If you want, you can also import previously saved TER files rather than re-creating height maps.

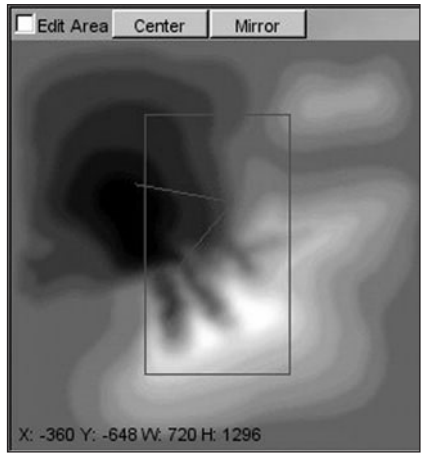


Figure 12.20 The mission area.

note

Reference to the newly created terrain file is stored in the mission file in a `TerrainBlock` that needs to be named "Terrain":

```
new TerrainBlock(Terrain) {
   rotation = "1 0 0 0";
   scale = "1 1 1";
   detailTexture = "~/data/terrains/details/detail1";
   terrainFile = "./myterrain.ter";
   squareSize = "8";
      locked = "true";
      position = "-1024 -1024 0";
};
```

Establishing Terrain Sizes

The units displayed in the Mission Editor map (x, y, w, h) represent the (x,y) distance of the upper-left corner of the mission area (in red) from the image center and the (w,h) width and height of the area in terrain texture units. Note that the position parameter in the mission file also uses the terrain texture units to position one terrain repetition. There are 32 repetitions of the terrain textures (don't confuse these with the height-map images) with each terrain texture image being 256 by 256 pixels.

32 reps×256 pixels=2,048 texture units
65,536 WU÷2,048 texture units=32 WU per texel

This information will be useful when creating terrain textures. Convert these values to inches by multiplying by 32. (The total area represented ranges from −1,024 to +1,024 when the terrain `squareSize`=8 for a total 2,048. And 2,048×32=65,536.)

If your contour area needs to be other than 1.034 miles, you can change the terrain `squareSize`. This will determine the area available before the terrain repeats. As you can see in Table 12.3, you must adjust the `squareSize` parameter in powers of 2.

Table 12.3 Terrain Sizes

Terrain SquareSize	Texels +/−, total	Texels × 32 = WU	Feet	Miles	Meters
32	+−4,096=8192	8,192×32=262,144	21,845.33	4.137	6,658.13
16	+−2,048=4,096	4,096×32=131,072	10,922.66	2.068	3,329.06
8	+−1,024=2,048	2,048×32=65,536	5,461.33	1.034	1,664.53
4	+−512=1,024	1,024×32=32,768	2,730.66	0.517	832.26
2	+−256=512	512×32=16,384	1,365.33	0.258	416.13

Team LRN

Changing the terrain squareSize in the mission file also affects the control in the Terrain Editor and terrain material painter; you will have more control at smaller sizes. Be sure to change the position values of the terrain to correspond to the worldSize also. For example, if you want more control of the terrain editing, set the squareSize to 4 and the position to −512 −512 0:

```
new TerrainBlock(Terrain) {
   rotation = "1 0 0 0";
   scale = "1 1 1";
   detailTexture = "~/data/terrains/details/detail1";
   terrainFile = "./myterrain.ter";
   squareSize = "4";
   locked = "true";
   position = "-512 -512 0";
};
```

Applying Terrain Cover

Terrain textures must be PNG format images and must be 256 by 256 pixels in size. These textures should be placed in a subdirectory under C:\3DGPAi1\fps\data\terrains; they will also work directly in the terrains folder.

Terrain textures are stretched to 2,048 world units (WU) if the terrain squareSize is 8. This means there are 32 repetitions of a terrain texture across one terrain width or depth (1 terrain rep). This also means there are 8 WU per texture pixel (texel).

65,536 WU÷32 texture reps=2,048 WU per texture rep
2,048 WU÷256 pixels=8 WU per texel

If the terrain squareSize is set to 4 in the mission file, there will still be 32 terrain texture repetitions, but each repetition will only cover 1,024 WU of the terrain.

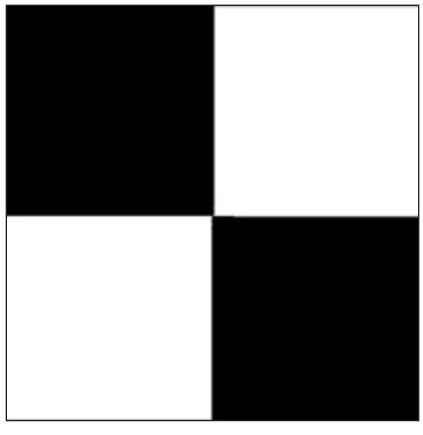


Figure 12.21 Checkerboard texture.

And although not a requirement, terrain cover textures will look best when they are created to be tiled, as discussed earlier; opposite edges should match so that when they are tiled you won't be able to see the edges. The images in Figure 12.21 and 12.22 are test textures that are 256 by 256 pixels. The checkerboard pattern in Figure 12.21 has each white or black section sized at 128 by 128 pixels.

The grid texture in Figure 12.22 has white lines every 32 pixels and red lines at 128 pixels. You can use these images to calculate the total terrain size with respect to the dimensions of objects created in

Team LRN

a map editor as well as to calculate the terrain square size with respect to terrain textures. In addition, you can use the image in Figure 12.22 to create sightlines when manually adjusting terrain heights.

To paint our terrain cover:

1. Place these images in a subdirectory under C:\3DGPAi1\fps\data\terrains.
2. Use the Run fps Demo shortcut to launch Torque.
3. Select the mission you created in the previous section.
4. Press the F8 function key to switch to "fly" mode.
5. Fly up above the terrain a bit using the arrow keys to move and the mouse to aim and look down. You can use F7 to switch out of fly mode when you want.
6. Press F11 to open the World and Terrain Editor.
7. Choose Window, Terrain Texture Painter, as shown in Figure 12.23.

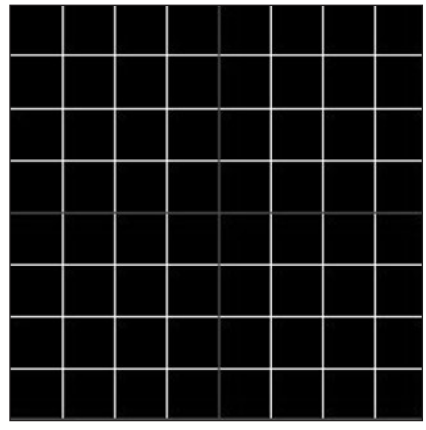


Figure 12.22 Grid texture.

You will now see the Material Selection dialog box (as shown in Figure 12.24) to the right. You can highlight the material you want to paint with or change or add new textures here.

8. To add a new material, click an Add or Change button and you will get a new texture image Load File dialog box, as shown in Figure 12.25.
9. Highlight the image you want and click the Load button. That image is now in your selection set.
10. From the Action pull-down menu, make sure Paint Material is checked.
11. Now go up to the Brush pull-down menu and select your desired brush size.

Figure 12.23 World Editor Window menu with Terrain Texture Painter checked.

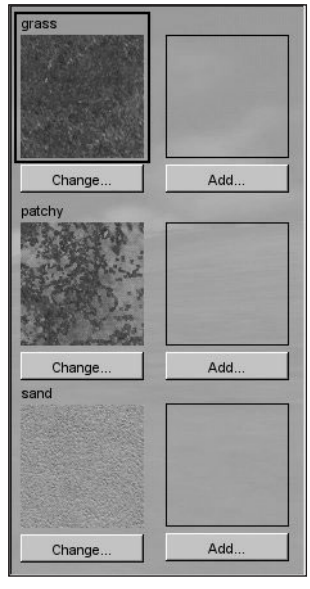


Figure 12.24 Material Selection dialog box.

Remember that we are using the default terrain `squareSize` set to 8. Table 12.4 lists the area of the terrain that is influenced based on brush size.

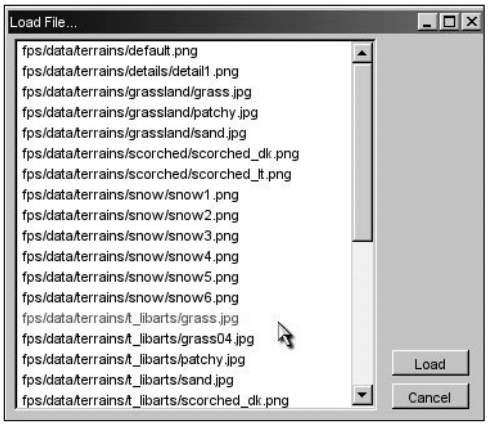


Figure 12.25 Image Load File dialog box.

Table 12.4 Brush Sizes

Brush Size	Texels	World Units (texels × squareSize)
1	1×32=32	32×8=256
3	3×32=96	96×8=768
5	5×32=160	160×8=1,280
9	9×32=288	288×8=2,304
15	15×32=480	480×8=3,840
25	25×32=800	800×8=6,400

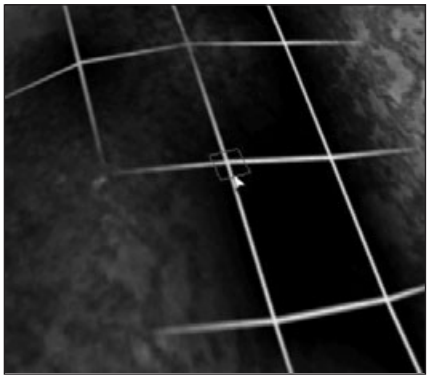


Figure 12.26 Painting Terrain with a brush size set to 1.

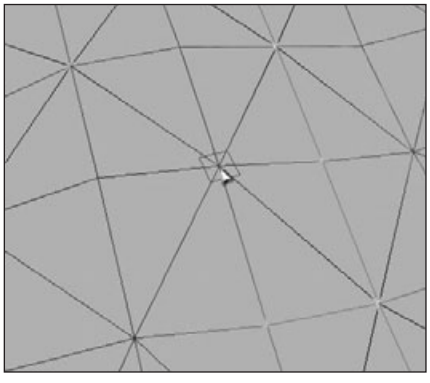


Figure 12.27 Terrain Grid with a brush size set to 1.

Figure 12.26 depicts a texture applied with the brush size set to 1, and Figure 12.27 shows the corresponding terrain grid.

You can see the 32 by 32 texel influence area for one brush that corresponds to 256 WU for the terrain squares shown in the grid view.

So take your trusty Terrain Paint Brush and go nuts!

Moving Right Along

So, now we understand why terrains need to be modeled, and what our options are for obtaining real-world terrain data. If we aren't modeling a real location, we've seen how we can create our own imaginary terrain using Paint Shop Pro, so that we can satisfy the needs of our game. We also looked at terrain cover, and how to create images for use as terrain cover.

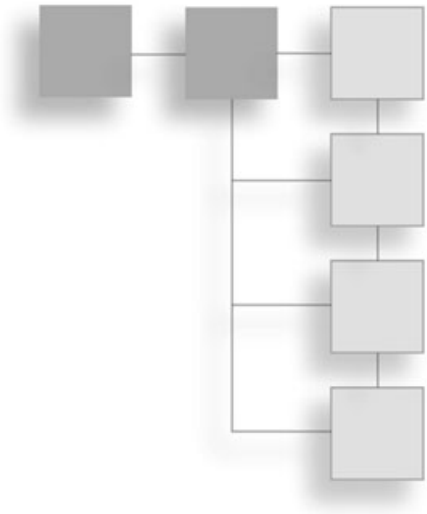
We also learned about some of the visual anomalies, like terrain tiling seams that might make our terrains less pleasing, and how we can go about fixing those issues.

Earlier in the chapter, Figure 12.8 showed an example of a finished terrain, with some hills in the distance, and terrain cover applied.

In the next chapter, we'll learn a pair of new tools, MilkShape and UV Mapper.

Team LRN

CHAPTER 13



Introduction to Modeling with MilkShape

In this and the following chapters, we will be delving into the world of low-poly modeling. We'll talk about techniques and methods that can be applied to other tools, such as the expensive 3D Max or Maya, but the practical focus will be geared toward using MilkShape, UVMapper, and other low-cost tools that are included on the enclosed CD.

MilkShape 3D

In Chapter 9 we created a skin for a simple soup can—remember that? Well, in this chapter we're going to create the model and skin it with the texture you created earlier, only this time we will go beyond just the simple soup can. But first, let's start at the beginning and learn a bit about MilkShape.

MilkShape 3D is a great low-cost low-poly 3D modeling tool created by a fellow named Mete Ciragan. Like most successful shareware applications, it has evolved over the years, as Mete added features requested by his user community. He also added the capability for users to create their own plug-ins to provide additional features and import-export filters.

MilkShape is not as complex as the more expensive tools, but that does not in any way imply that it is not a capable program, especially in the low-poly world that computer games inhabit. In fact, the stripped-down nature of MilkShape certainly makes it easier to learn than most of the "big boys."

Installing MilkShape 3D

If you want to install only MilkShape 3D from the enclosed CD, do the following:

1. Browse to your CD in the \MS3D directory. (MS3D is the abbreviated form for MilkShape 3D. You'll encounter it a fair bit.)

Team LRN

2. Locate the Setup.exe file and double-click it to run it.
3. Click the Next button for the Welcome screen.
4. Follow the various screens and take the default options for each one, unless you know you have a specific reason to do otherwise.

note

MilkShape does not know how to open its own files when you double-click them to be launched from Windows Explorer. You need to launch MilkShape first and then browse for your files with the MilkShape File, Open dialog box.

The MilkShape 3D GUI

tip

If you only have three views in your window when you first run MilkShape, choose Window, Viewports, 4 Window, and you should get something close to what you see in Figure 13.1.

In Figure 13.1 there are four places where the model can be seen. Each of these is what MilkShape calls a *window*. We will call them *frames* in this book, because as you probably already know, MilkShape itself is in a window.

A *view* is the angle or direction at which you look at an object. For example, if you stand in front of an object and look at it, you are seeing the *front view*. From above, it is the *top view*.

A *viewport* is the little frame inside the MilkShape window in which a view of a model is presented. When I want to direct your attention to a particular viewport, I might refer to it as being in a particular *frame*. MilkShape calls these frames *windows*, which is a bit of a misnomer.

For example, in Figure 13.1 the 3D *view* is in the 3D *viewport*, located in the lower-right *frame* in the MilkShape *window*.

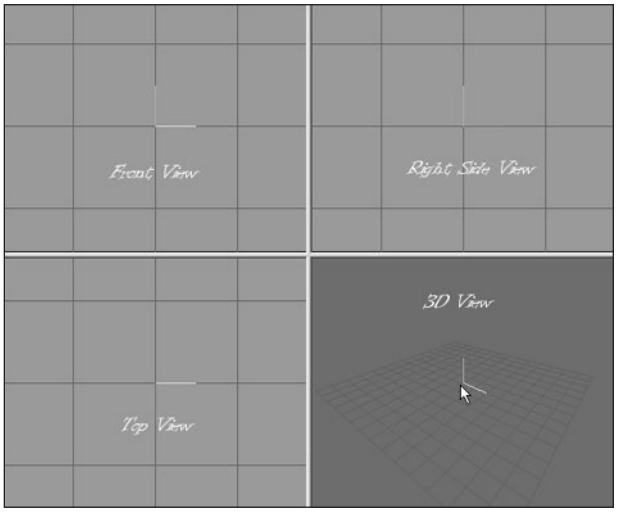


Figure 13.1 MilkShape 3D.

You'll notice in Figure 13.1 that I've labeled the different views. This is the way you should use your views for models that you create for Torque. Other applications and games may require your models to be oriented differently.

Three of the views are wire-frame-only views; they allow you to look at your model from directly above, directly in front, and the right-hand side. The fourth view is a 3D view in which you can rotate your model various different ways and view it as a wire-frame, shaded, or fully textured model with lighting cues.

Figure 13.2 shows the tools available in the toolbox section. Although some tools for different operations are only available in the menus, most of the time you will be working with the tools in the toolbox.

Navigating in Views

In the wire-frame views, you can move the view around by holding down the Ctrl key and clicking and dragging in the window.

If you hold down the Shift key and drag the mouse, you can zoom in or out. Be careful though—if you are in select mode (from the Model tab), the program will alternate between zooming and object-selecting each time you press the Shift key and drag the mouse button. With practice you can master this and it will become quite useful. This behavior only occurs in select mode. In all other modes, the shift-drag action will always zoom the view in or out with no alternation.

If you have a wheel mouse, then the wheel can be used to zoom in or out. You will have to click in the view to get focus into the view before the zoom will work.

The 3D view allows the view movement in the same ways as the other views, except the wheel mouse zoom works backward.

View Scale and Orientation

When you are viewing an object from the front in MilkShape, the Y-axis is positive going up, the X-axis is positive going to the right, and the Z-axis is positive going to the front. This makes it a right-handed coordinate system.

If you look at the Right Side view (the view at the upper right of the four), you will see in the center the axis "bugs" for the Y- and Z-axes. Although it is not visible in the black-and-white

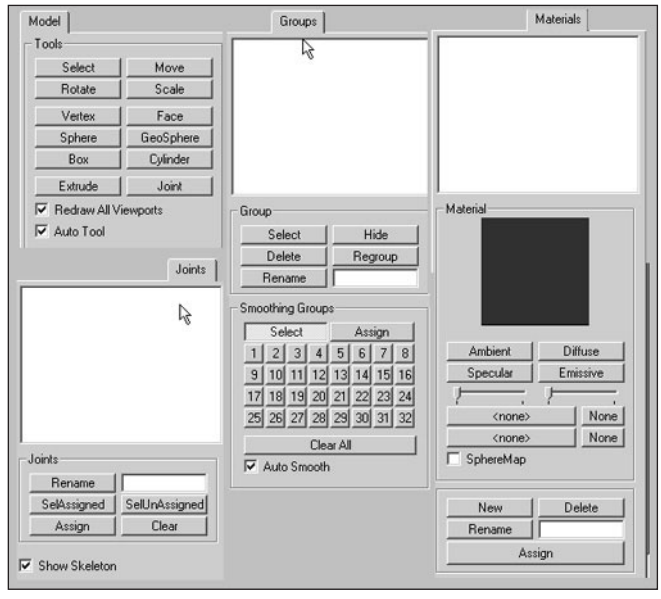


Figure 13.2 The toolbox contents.

pictures in this book, the Y-axis line is cyan and the Z-axis line is magenta. The place where these two lines meet is the (0,0,0) coordinate in object space. Hold your mouse cursor over the first grid line above the (0,0,0) location and look down to the lower-left corner of the MilkShape window while keeping your cursor over that grid line. You should see the Y-axis value at about 20.0 or so (see Figure 13.3). If you see 20.005 or 19.885, that's good enough. If you don't see 20.0 or so, zoom the view in or out until you do. Adjust your other two wireframe views to the same scale. If you position your cursor one grid line directly above the (0,0,0) point on the Front view (upper left), you should see the 20.0 or so also for the Y value, but for the Top view, the same relative positioning will be affected in the Z-axis.

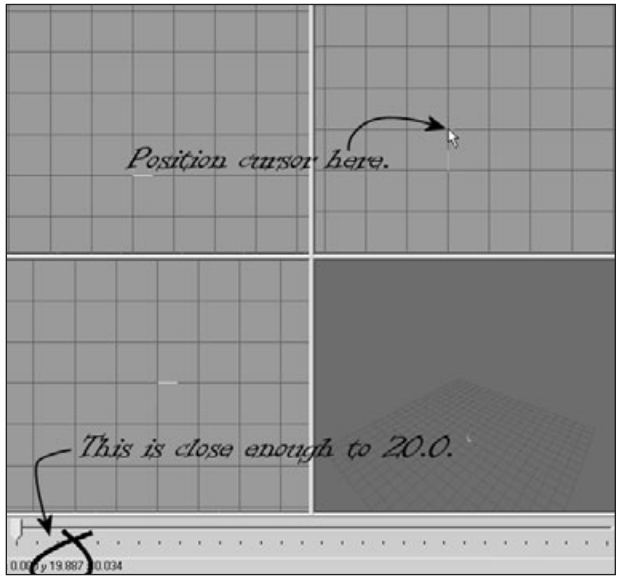


Figure 13.3 Checking the zoom in the Right Side view.

Figure 13.4 contains various notations to help you understand the coordinate display system. In this figure, I've left in MilkShape's viewport labels above each viewport's frame in order to illustrate the variation that emerges with the Torque Right Side view being seen in MilkShape's Left viewport.

The whole point of this little exercise is to expose you to the coordinate display and to ensure that your layout matches the one we'll be working with here. Of course, at times when you zoom in and out this might change, but now you have a method of recalibrating when necessary.

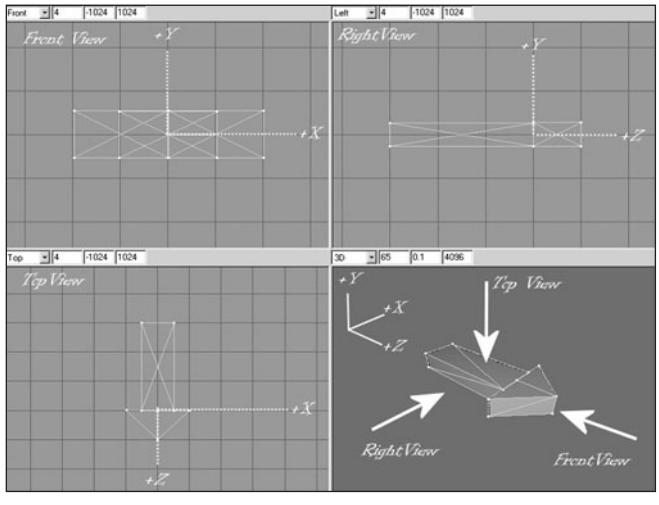


Figure 13.4 Torque-oriented object in the MilkShape viewports.

The Soup Can Revisited

Now that you have a bit of a grasp of what you are looking at in the GUI, and how to move your views around to look at your model, we'll move on to actually creating a quick model to get your feet wet. There's nothing like doing for learning!

Creating the Basic Shape

A closed can is a cylinder. A cylinder is what we call a *primitive* shape, like a sphere or a cube. The primitives are added together in various ways to build up more complex shapes.

1. Choose the Model tab in the toolbox.
2. Click on Cylinder.
3. Position your cursor in the Right Side view about three grid lines above (0,0,0) and one grid line to the left. Click and drag down and to the right until your object looks like Figure 13.5.
4. Choose the Groups tab. You will see a single group named cylinder01 (it may be a higher number if you made more cylinders and deleted them—MilkShape just adds 1 to the number at the end during its auto-naming).

tip

You may recall encountering the term *mesh* way back in Chapter 3. In MilkShape the word *group* is actually an analogue for the word *mesh*. They mean essentially the same thing.

5. Click on the group name to highlight it, and then type **can** in the box to the right of the Rename button where it says "Cylinder01".
6. Click Rename, and the group will now be called can.
7. Choose the Materials tab.
8. Click New.
9. Type **label** into the Materials Rename box.
10. Click Rename.
11. In the Material frame of the Materials tab you will see two buttons labeled "<none>". These are the texture buttons. The top one assigns the standard texture, and the bottom one allows you to assign a texture whose alpha channel you want to use for this material.
12. Click the top texture button. You will get a file dialog box.

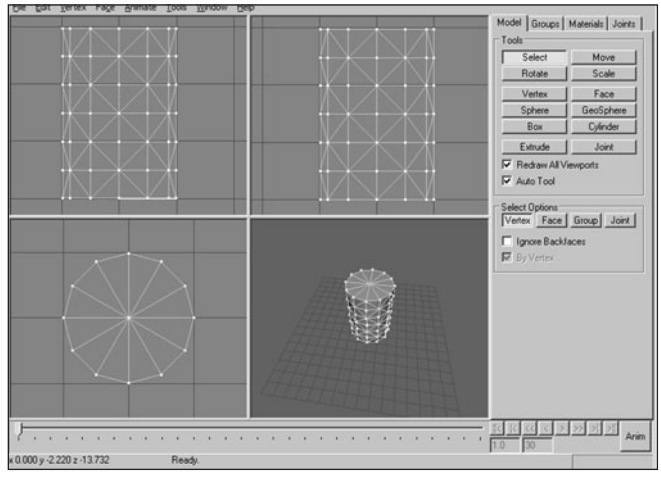


Figure 13.5 Making a cylinder.

13. Browse your way to C:\3DGPAi1\resources\ch9 and double-click the can.jpg file.
14. Now choose the Groups tab again, make sure your cylinder's group is selected in the list. If your can is not already highlighted in red, click Select. You will see your can highlighted in red in the three wire-frame views.
15. While your can is still selected, switch back to the Materials tab, choose your new material in the list, and click Assign.

tip

If your screen resolution is set to 800×600 or less, you will not be able to see the entire Assign or Select By buttons. The top one-quarter or so of those buttons is just visible on the bottom-right corner. Assign is located below the Rename button, and Select By is located below the edit box that is to the right of the Rename button.

16. Right-click in the 3D view and choose Textured. Your can should appear with the texture wrapped around it, like in Figure 13.6.
17. Save your work so far as mynewcan.ms3d somewhere, by choosing File, Save As.
18. In preparation for UV unwrapping the can object, choose File, Export, Wavefront Obj and export the file to C:\3DGPAi1\resources\ch9\mynewcan.obj.

Okay, so we have the soup can made, and we've assigned the texture to it. The reason the texture doesn't fit right yet is because the texture coordinates haven't been mapped to the object yet. That's our next step.

UV Unwrapping the Can

In Chapter 9 we encountered some of the theory and process behind UV unwrapping and mapping. In a later section in this chapter we'll go into more theory, as well as more detail about the UVMapper tool. For our purposes at the moment, we just want to get the texture skin mapped correctly onto the can.

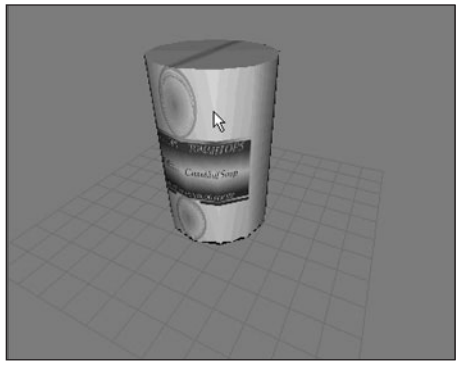


Figure 13.6 Assigned texture.

Whether the skins are created before the object or the object is created first will probably change from project to project or even from phase to phase within a project. At this point in the book, we already have a skin—can.jpg—so we want to make sure the can will unwrap to match the skin. This isn't a problem in this case. It may be a problem with other projects though, so be aware of that possibility.

1. Using Windows Explorer, browse your way to C:\3DGPAi1\resources\tools and launch UVMapper.exe.
2. Maximize the window when it opens.
3. Find the file you exported, C:\3DGPAi1\resources\ch9\mynewcan.obj, and open it.
4. You will see an alert, listing some statistics about the object. Click OK.
5. You will see a bunch of triangles fill your window. Ignore them for the moment.
6. Choose Edit, New UV Map, Cylindrical Cap. You will get a Cylindrical Cap Mapping dialog box.
7. Click OK. You will then get a layout of the can's triangles (like that in Figure 13.7), with a rectangular block of triangles across the middle and a circle of triangles at both top and bottom.
8. Choose File, Save Model. The OBJ Export Options dialog box then appears.
9. Set the options boxes as shown in Table 13.1 and click OK.
10. Replace the OBJ file C:\3DGPAi1\resources\ch9\mynewcan.obj by saving over it.
11. Choose File, Save Texture Map. The BMP Export Options dialog box appears.
12. Set the options to the values shown in Table 13.2.
13. Save to the file name C:\3DGPAi1\resources\ch9\mynewcan.bmp. This is the texture map, or UV mapping template, for your can.
14. Switch back to MilkShape.
15. Choose the Groups tab and select the can group.
16. Click the Delete button. You will replace this object with the one you exported from UVMapper.
17. Choose File, Import, Wavefront Obj and import the mynewcan.obj file you saved from UVMapper.
18. On the Groups tab, locate your new object (mynewcan.obj), select it, and rename it if you like.
19. With the new object selected, choose the Materials tab.
20. Choose the label material and then click Assign.

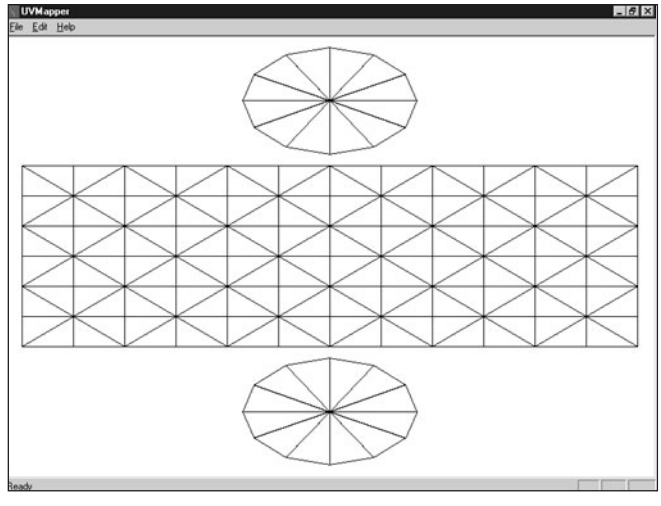


Figure 13.7 Unwrapping the can in UVMapper.

Table 13.1 UVMapper OBJ Export Options Values

Value	Option
clear	Export As Single Group
set	Export Normals
set	Export UV Coordinates
clear	Flip Texture (UV) Coordinates Vertically
clear	Flip Texture (UV) Coordinates Horizontally
clear	Reverse Winding Order
clear	Invert Normals
clear	Swap Coordinates Y and Z
set	Export Materials
set	Export UVMapper Regions
clear	Export Using Rotation Settings
clear	Don't Export Linefeeds (Mac compatible)
clear	Don't Compress Texture Coordinates

Table 13.2 UVMapper BMP Export Options Values

Value	Option
512	Bitmap Size—Width
512	Bitmap Size—Height
clear	Flip Texture Map Vertically
clear	Flip Texture Map Horizontally
clear	Exclude Hidden Facets

21. Your texture should appear on the can in the 3D view, wrapped correctly.
22. If no texture appears, click in the 3D window to force an update.
23. If there is still no texture, make sure that you have the 3D window still set to Textured, by right-clicking in the 3D window and checking the menu.

Enhancing the Soup Can Model

Have a seat and stew on that for a while. When you are done, we'll carry on and start hammering at the soup can and improve the model.

How about we open the can up? The can model has a top and a bottom. We want to leave the bottom where it is and flip the top lid up.

First we need to separate the lid from the can.

1. Choose the Model tab and click Select.

2. Click Vertex and select all the vertices at the bottom of the can, as shown in Figure 13.8. Use either the Side view or the Front view.
3. Choose Edit, Hide Selection. The dots of the vertices will disappear. This means that none of the vertices for the bottom face are selectable.
4. Now click Face. Make sure that By Vertex is selected.
5. In the Top view, select the vertex in the center of the can, as in Figure 13.9. Because you had hidden the bottom vertices, only the single center vertex for the top of the can has been selected. And because you are actually selecting faces by vertex, then all the top lid faces—and only those faces—have been selected.
6. In the Groups tab, click Regroup. This will create a new mesh with only the faces from the top of the can. The mesh will be named "Regroup01". Rename this mesh to "lid" in the same manner that you did earlier when you renamed the cylinder mesh to "can".
7. Switch back to the Model tab. Your lid mesh should still be selected.
8. Click the Move button, and then click and drag in the Side view to move the lid up and to one side from the rest of the can.
9. Click the Rotate button, and then click and drag in the side window to rotate the lid as if you were bending the lid back (see Figure 13.10).
10. If necessary, repeat step 8 to position the lid properly. You might have to adjust it in one of the other views, depending on how you initially moved the lid.

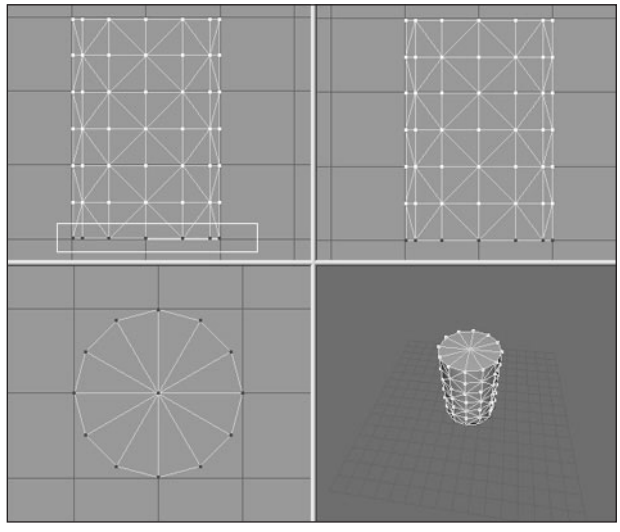


Figure 13.8 Selecting the bottom vertices.

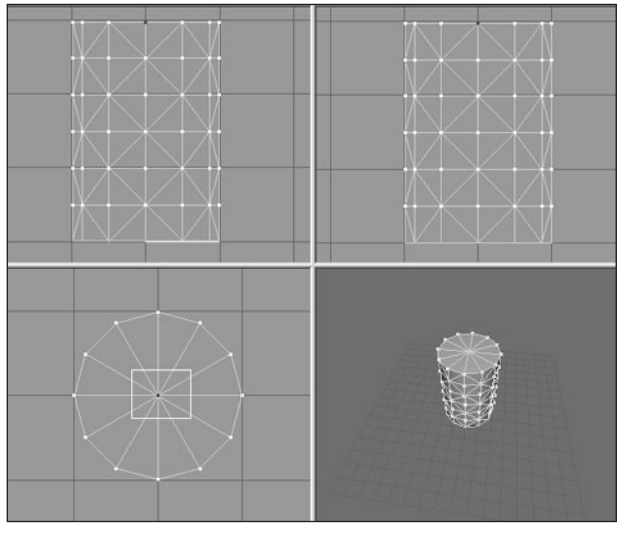


Figure 13.9 Selecting the center vertex.

Team LRN

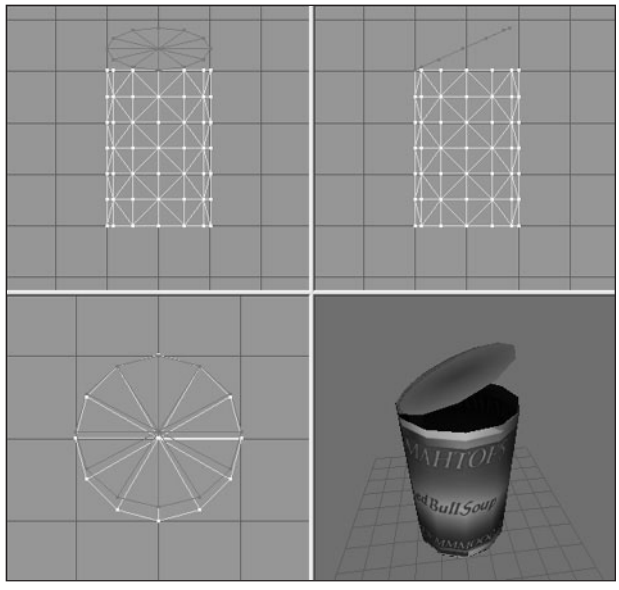


Figure 13.10 The can with lid opened.

11. Choose the Groups tab and click Regroup. The lid faces will now be part of their own group.
12. Choose Edit, Duplicate Selection. Another copy of the lid will be made in exactly the same location as the original.
13. Select Face, Reverse Vertex Order. This will invert the normals of the lid's faces, making it viewable from the other direction. You will recall that the normal of a face is, in the simplest terms, the direction that a face is facing.
14. On the Groups tab, add the original lid to your selection by clicking on the first lid group.
15. Select Vertex, Weld Together. Now the original lid is viewable from one side, and the copy is viewable from the other. They now share the exact same vertices.

    If you rotate your can in the 3D view, you'll see that your lid now has the lid part of the skin on both sides. You'll also notice that the inside of the can is black. This is because no faces are normalized to the interior, just as the lid at first did not have any faces normalized on the side.

tip

You may be wondering why you didn't have to assign a material to the new faces you created with the Duplicate command. What happened is that when you grouped the original faces and the new faces together, the material that was assigned to the original lid faces were automatically assigned to the new group.

16. Repeat the above steps but this time create a set of faces for the can body that are normalized to the interior instead of the exterior, and then group them together. You can use your UV mapping and Paint Shop Pro skills to create a more realistic metallic interior to the can, instead of just repeating the exterior skin on the inside.
17. Save your work—you never know when a nice can of soup may be needed for dipping your towel in!

So, here we are. You've made a model of an object, using a couple of shape primitives. And you've learned how to make double-sided textures, rotate and move meshes (or groups), and assign skins. Feel free to explore your new capabilities. Poke around and try out the other primitives.

Menus

MilkShape can perform many more features and operations than what we've just gone over. In later chapters you'll learn how to make more difficult and challenging shapes, like player-characters, vehicles, and weapons. In this chapter we'll take a look at the program itself in more detail.

Most but not all of the menus have shortcuts assigned to the keys. Typically, the ones that are used the most do have shortcuts. If you want to add your own shortcut, you can use a plug-in to do that. We'll cover that when we discuss the Tools menu.

File

As in most Windows programs, operations in the File menu (see Figure 13.11) relate either to the creation and saving of files or to making global alterations to the current file's properties or contents. See Table 13.3 for more detail.

Figure 13.11 The File menu.

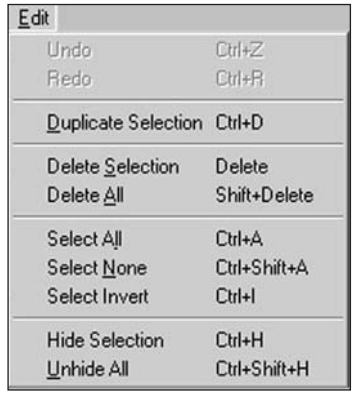


Figure 13.12 The Edit menu.

Edit

The MilkShape Edit menu (see Figure 13.12) contains commands that assist the user when modifying models. It does not have Cut, Copy, or Paste but does offer commands in a similar vein for duplicating, hiding, and selecting objects. See Table 13.4 for more detail.

Vertex

You can perform a number of operations on vertices in a model. They are available through the Vertex menu (see Figure 13.13). In most cases you will need to ensure that you've selected only vertices in a model or at least have the selection mode set to Vertex. See Table 13.5 for more detail.

Figure 13.13 The Vertex menu.

Table 13.3 MilkShape File Menu

Command	Description
New	Creates a new blank workspace. If the current workspace is not empty, then the user is prompted to save changes or continue without saving. Only one workspace can be open at a time.
Open	Opens an existing MS3D formatted file using the standard Open dialog box.
Save	Saves the current workspace as an MS3D file, providing that the current workspace has a name. If the workspace is unnamed, then the command will behave like Save As.
Save As	Requires the user to specify a new file name under which to save the workspace contents.
Merge	Merges together two MS3D documents: the current workspace and another workspace selected from file.
Import	Presents a submenu of file import plug-ins. This command works like the Open command, once an import plug-in is selected, except that some plug-ins offer import options in a user dialog box.
Export	Presents a submenu of file export plug-ins. This command works like the Open command, once an import plug-in is selected, except that some plug-ins offer additional export-specific options in a user dialog box.
Preferences	Presents the Preferences dialog box. This allows the user to set definable global application attributes and behaviors.
Exit	Exits the MilkShape program. The user is prompted to save changes if there are any that haven't been saved.

Table 13.4 MilkShape Edit Menu

Command	Description
Undo	The workspace is reverted back to the state it was in before the last user operation.
Redo	The workspace is reverted back to the state it was in before the Undo operation.
Duplicate Selection	All selected objects are duplicated in place. This command selects the new duplicates and deselects the previously selected objects.
Delete Selection	The currently selected objects are deleted.
Delete All	All objects in the workspace regardless of their selection state are deleted.
Select All	All objects in the workspace are selected.
Select None	All objects in the workspace are deselected.
Select Invert	All objects that were selected are deselected, and all unselected objects are selected.
Hide Selection	The selected object is hidden from view. This operation can also be performed on groups using the Groups tab in the toolbox.
Unhide All	All objects in the workspace are shown.

Team LRN

Table 13.5 MilkShape Vertex Menu

Command	Description
Snap Together	Snaps all the selected vertices together. The middle point between all selected vertices becomes the new location for the vertices.
Snap To Grid	Moves all selected vertices to be in line with the smallest grid X, Y, and Z position (to see the smallest grid positions, zoom all the way in). The grid size can be changed using the File, Preferences menu.
Weld Together	Creates one vertex at a precise point where several vertices exist. Only selected vertices are welded together. This is the way you would join a seam of two or more abutting faces.
Unweld	Splits each selected vertex into multiple vertices. The number of vertices created depends on the number of faces the original individual vertices were bound to. For example, a vertex with three faces attached will be split into three vertices.
Unweld Radial	Is the same as Unweld but will also shift the unwelded vertices away from each other in a circular pattern. The vertices will move from the origin at which they were unwelded by half the distance from the origin to the nearest edge.
Divide Edge	Divides a face between two selected vertices into two faces. The procedure will only work with two vertices selected. This has no effect on vertices without any faces in common.
Flatten	Presents a submenu for the user to align all selected vertices to the same point on the X, Y, or Z plane. This is similar to Snap Together but it works on only one axis instead of all three.
Mirror Front <--> Back	Mirrors, or flips, the currently selected object along the Z-axis.
Mirror Left <--> Right	Mirrors, or flips, the currently selected object along the X-axis.
Mirror Top <--> Bottom	Mirrors, or flips, the currently selected object along the Y-axis.
Spherify	Calculates a bounding sphere and attempts to place the selected vertices on the surface of the sphere. It can be constrained in all three dimensions, and the bounds can be manually set.
Manual Edit	Allows the exact placement of one selected vertex with floating point accuracy in the X, Y, and Z planes.

Face

The Face menu (see Figure 13.14) provides commands for manipulating triangles and faces in the workspace. See Table 13.6 for more detail.

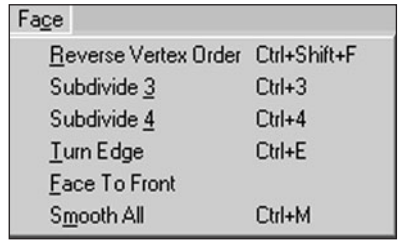


Figure 13.14 The Face menu.

Table 13.6 MilkShape Face Menu

Command	Description
Reverse Vertex Order	Changes the order of the vertex winding, which changes (negates) the normal of the face. This will turn a face inside or outside depending on its current vertex order. Counterclockwise vertex winding creates an outward face on an object.
Subdivide 3	Divides each selected face by three, creating three faces out of one.
Subdivide 4	Divides each selected face by four, creating four faces out of one.
Turn Edge	Is performed on two triangles with a common edge. The common edge is removed, and a new edge is created between the two vertices (one from each triangle) that weren't previously joined by an edge.
Face To Front	Is used on selected faces to change all vertex orders to counterclockwise, outward-facing vertex ordering, or *winding*.
Smooth All	Is used to smooth all faces after construction of parts of a model.

Animate

The Animate menu (see Figure 13.15) is used to manipulate animation frames in the model via the Keyframer. See Table 13.7 for more detail.

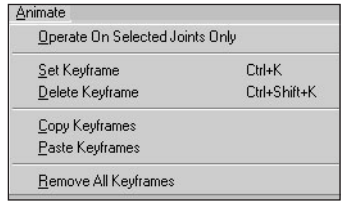


Figure 13.15 The Animate menu.

Table 13.7 MilkShape Animate Menu

Command	Description
Operate On Selected Joints Only	When this menu item is toggled on (checked), then only the joints that are currently selected will have their pose data stored for the current keyframe.
Set Keyframe	This stores the pose of the skeleton to the current keyframe (whichever keyframe that's in the keyframe number box).
Delete Keyframe	This removes the stored skeleton pose from the current keyframe.
Copy Keyframes	This copies the skeleton pose from the current keyframe. In order for the copy action to perform correctly, the user must first select the skeleton in the keyframe to be copied from.
Paste Keyframes	This pastes the copied skeleton pose to the current keyframe. After the keyframe has been pasted, you need to immediately set the keyframe in order to preserve the skeleton pose.
Remove All Keyframes	This removes all stored skeleton poses at all keyframes in the animation timeline. This is effectively the same as deleting the animation.

Team LRN

Tools

The Tools menu (see Figure 13.16) provides access to both built-in tools and user plug-in tools. The functions available are not the same as those available in the toolbox. This is a potential source of confusion. See Table 13.8 for more detail about the Tools menu.

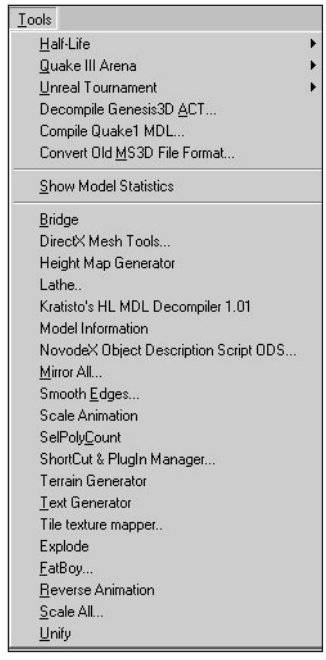


Figure 13.16 The Tools menu.

Table 13.8 MilkShape Tools Menu

Command	Description
Half-Life	This command contains several options used to create and save Half-Life models.
Quake III Arena	This command saves a Quake III control file to the directory you specify in the save window.
Unreal Tournament	This command contains options for creating male and female skeletons using the default Unreal Tournament skeleton configurations.
Decompile Genesis-3D ACT	This command allows you to decompile an ACT model used by the Genesis 3D engine.
Compile Quake1 MDL	This option will compile a Quake1 MDL file, used in the Quake 1 engine.
Convert Old MS3D File Format	This command converts a file from an old version of the MilkShape 3D file File format. The command can be used to change files created with MilkShape 3D versions 1.4.0 and earlier.
Show Model Statistics	Brings up a statistics window showing useful statistics, such as the number of faces and vertices in the workspace.
Plug-Ins	The list of plug-ins available is user-configurable using the Shortcut and Plug-In Manager. Not all plug-ins are distributed with MilkShape. See the MilkShape sidebar for descriptions of currently available plug-ins. To get the most up-to-date information about plug-ins, visit chUmbaLum sOft's Web site: http://www.swissquake.ch/chumbalum-soft.

Team LRN

MilkShape Plug-Ins

There is quite a large list of MilkShape plug-ins that extend MilkShape's capabilities. For information about where to find them to download, tutorials about how to use them, and the names of the individual creators, see Appendix B, "Game Development Resources on the Internet." The plug-ins that were known at the time of this writing are listed; some plug-ins are import or export filters for different file formats and aren't included here, except for the Torque DTS Exporter, because we use it in this book (the Wavefront OBJ Importer and Exporter are built into MilkShape).

- **ms2DTSExporter.** This plug-in exports models, animations, and materials to DTS model format for use with the Torque Engine. This plug-in appears in the File, Export menu.
- **msSelectionEditor.** This plug-in edits the selection from a 3D view. There are a lot of options and you can read some detailed information about it here.
- **msTimer.** This plug-in lets you time how long you've been working on a certain model.
- **msEdgeExtrude.** This plug-in lets you extrude edges in addition to faces.
- **msJointTool.** This plug-in allows you to add joints in the middle of the hierarchy, unlink a joint from a hierarchy, and assign vertices to the closest joint (some kind of "Assign Mesh to Skeleton" tool).
- **msSnap.** This plug-in snaps not only to 1.0, it also snaps joints.
- **msToolArray.** This plug-in duplicates objects and then places the duplicates in 3D space according to user specifications.
- **msVertexPlane.** This plug-in is similar to the Vertex, Flatten command, except that it snaps selected vertices to a plane instead of a single point.
- **msToolFatBoy.** This plug-in will make your model fatter or thinner. This is useful for tweaking player and monster characters.
- **msOperationMirrorAll.** This plug-in will mirror everything about your model over the selected plane: bones, mesh, animation—everything.
- **msToolReverseAnimation.** This plug-in will reverse the order of the keyframes in whatever animation you have loaded.
- **msToolScaleAll.** This plug-in applies scale to all objects in the workspace at once.
- **msSelPolyCount.** This plug-in shows the selected polygon, vertex, and unique vertex counts, as well as how many polygons there are per group.
- **msBridge.** This plug-in creates a mesh connecting to previously independent meshes or groups.
- **msTerGen.** This plug-in can generate random terrains or import a bitmap file to use as a height map.
- **msTextGen.** This plug-in generates 3D objects in the form of text.
- **msModelInfo.** This plug-in provides more detailed information about a model than the Show Model Statistics command.

Team LRN

- **msTileTextureMapper.** This plug-in generates texture coordinates to geometry for tile textures (also known as *seamless textures*).
- **msLathe.** This plug-in takes flat geometry and turns it around the X-axis to build a 3D model.

You can install plug-ins by simply copying them to the MilkShape directory and then launching MilkShape. They will then appear under the Tools menu beneath Show Model Statistics.

Window

The Window menu (see Figure 13.17) provides commands that determine what information is available in the MilkShape window and how it is displayed. See Table 13.9 for more detail.

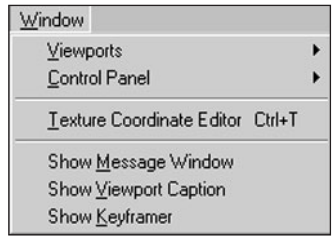


Figure 13.17 The Window menu.

Table 13.9 MilkShape Window Menu

Command	Description
Viewports	This command presents a submenu that allows you to pick an alternative viewport layout. The four-pane layout with three 2D views and one Perspective view is the default.
Control Panel	This command allows the user to set whether the toolbox frames appear on the left or right of the main window. The right side is the default.
Texture Coordinate Editor	This is for adjusting where textures appear on the model. Although useful, it is not as powerful or flexible as using a dedicated UV Unwrapping or Mapping tool like UVMapper.
Show Message Window	This option shows a script output window that holds the results of compiling various types of models for specific games.
Show Viewport Caption	This command shows details about the viewport it appears above. From left to right, the details are the view, the field of view, the near clipping plane, and the far clipping plane.
Show Keyframer	The Keyframer is the animation box along the bottom of the main window. It is used to create keyframe positions of bones and joints in a skeleton for animation.

The Toolbox

Way back near the start of this chapter, in Figure 13.2, is a depiction of the contents of the various tabs in the toolbox. In this section here we will dig deeper into the capabilities in those tabs. Table 13.10 provides a brief summary of each toolbox tab's functions.

Table 13.10 MilkShape Toolbox Summary

Tab	Purpose
Model	This is used for the placement of vertices and shape primitives, as well as for the construction of polygons and skeletons.
Groups	This contains commands used to group vertices and polygons. Groups can also be created from existing polygons.
Materials	This deals with the creation of materials, including textures from file, ready to be assigned to groups.
Joints	This contains tools for manipulating and managing skeleton joints.

You should understand that, in general, when using the functions in the toolbox, we will first have to select some object via one of the views and then operate on it using one of the toolbox commands. This sort of noun-verb operation mode requires us to make sure we have the appropriate objects selected before every action we take.

Figure 13.18 The Model tab.

The Model Tab

The Model tab (see Figure 13.18) contains the tools necessary to create and modify the basic shape primitives: vertices, faces, cylinders, spheres, and cubes (boxes, as MilkShape calls them). Table 13.11 shows the functions of the Model tab's buttons.

Table 13.11 Model Tab Functionality

Button	Description
Select	This tool puts the program into select mode so that the user can select any object or collection of objects on one of the wire-frame views. Once you are in select mode, you can specify one of four different selection target types: vertex, face, group, or joint. You also have two optional settings: ignore backfaces and by vertex (which is only available in face selection mode).
Move	This tool permits you to move any selected objects by clicking in the appropriate wire-frame view and dragging the cursor. You can also specify discrete movement by entering numbers in the Move Options boxes at the bottom.
Rotate	This tool permits you to move and rotate any selected objects around a single axis by clicking in the appropriate wire-frame view and dragging the cursor up or down. You can also specify discrete rotations and multiple axes rotations by entering numbers in the Rotate Options boxes at the bottom.
Scale	With this tool you can change the size of any selected objects along one of two available axes in each view by clicking in the appropriate wire-frame view and dragging the cursor up, down, left, or right. You can also specify discrete scaling and multiple axes scaling by entering numbers in the Scale Options boxes at the bottom.
Vertex	Use this tool to place individual vertices, one at a time, in a wire-frame view. In each different view, the vertex will be placed at the zero axis position for whichever axis is not presented in the view. This tool has no options.
Face	With this tool you can connect individual vertices to create a face, one vertex at a time, with three vertices defining a face. The Threshold option specifies how close to a vertex you need to click to add it to the current face you are building. As you build the face, select the vertices in a counterclockwise direction to create an outward normalized face.
Sphere	This tool is a shape primitive tool. To create a sphere, simply click and drag the cursor in a wire-frame view. With the Sphere options you can specify the number of slices (like the slices in a pie) or stacks (like a stack of pancakes) that make up your sphere.
GeoSphere	Use this tool to create more realistic spheres using a different program technique. You use the tool the same way as the Sphere tool, but you can only specify the complexity of the sphere using the Depth option.
Box	Use this tool to create cubes. Just click in a wire-frame view and drag until it has reached the size you want.
Cylinder	Use this tool the same way as you use the Sphere tool, even including the specification of stacks and slices.
Extrude	This tool operates only on the faces. If you have two faces aligned to create a flat surface, like a piece of cardboard, you would use this tool to extend the surface in a specific direction to create a box. Just click the mouse and drag to perform the extrusion. Using the Extrude options you can specify which directions to do the extruding in—normally you would use only one direction at a time. The Smoothing option tells the program to smooth shade the polygons as it draws the extruded shape.

continued

Joint	This tool places special joint objects. It works the same as the Vertex tool, except that if an existing joint is already selected when you make the new joint, the new joint will be attached to the previous one by a bone. If the Show Skeleton option is turned on in the Joint tab, the bone will be visible in yellow.
Redraw All Viewports	If you have this option turned on, then every time you perform one of the tool operations, the views in all the viewports will be redrawn to reflect your changes.
Auto Tool	If you have this option turned on, then the program will alternate between any tool and the Select tool each time you finish an operation. This option is handy for tweaking and repetitive adjustment techniques.

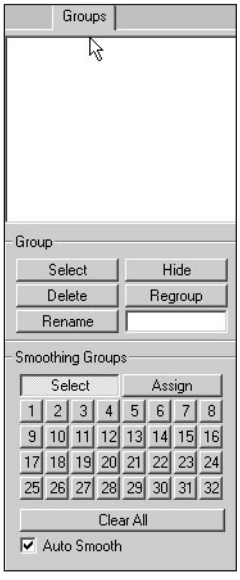


Figure 13.19 The Groups tab.

The Groups Tab

You will often want, or need, to organize your model faces into groupings that make either visual or logical sense. Whether you organize them as meshes that make visual sense or simply as logical groups, you do this with the Groups tab, shown in Figure 13.19. The Torque DTS Exporter uses special groups with the name *collision* to define collision meshes. Table 13.12 presents the functions available from the Groups tab.

The Materials Tab

With the Materials tab (see Figure 13.20), you can define the textures that will be used to skin your model, as well as what characteristics they will have when displayed. Special materials are also used to define certain model characteristics to the Torque DTS Exporter. Table 13.13 explains the functions of the Materials tab.

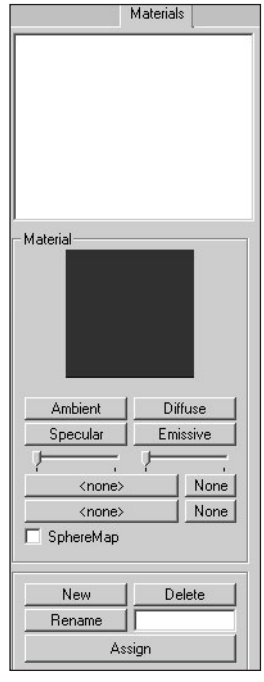


Figure 13.20 The Materials tab.

Table 13.12 Groups Tab Functionality

Button	Description
Group Selector Box	This is the white area at the top. It contains the names of the groups, one group per line. You always need to choose a group from this box before performing any group operations.
Select	When you use this tool, the currently chosen group in the list will become selected in the wire-frame views—that is, it will become drawn in red. Each time you choose a different group and click the Select tool, that group gets added to each view's selection.
Hide	With this tool you can make the chosen group's faces and vertices become invisible. This is useful for uncluttering a view or to ensure that you don't select the wrong parts for another operation.
Delete	Use this tool to permanently remove a group from the model.
Regroup	With this tool you create new groupings from whatever model elements are selected (shown in red) in the views. Any elements that already belong to other groups are removed from those groups and added to the new group.
Rename	Choose a group, type a new name in the Rename box, and then click the Rename tool. Bingo! The group now has a new name.
Smoothing Groups Select	Pressing this key and then one of the Smoothing Group numbers selects the polygons assigned to that Smoothing Group. Smoothing Groups can only be selected on the numbers that have been assigned.
Smoothing Groups Assign	When you have a group of polygons selected you can press this button and then one of the Smoothing Group numbers to assign all selected polygons to a Smoothing Group. Additional groups of polygons can be added to the same group without overwriting the previous contents of the Smoothing Group.
Smoothing Group Numbers	These numbers act as a storage bank for groups of polygons. They can have polygons assigned to them and selected from. If the Auto Smooth check box is selected, assigning groups of polygons to a Smoothing Group number will smooth shade them (Smooth Shaded shading has to be enabled to view the effect of this—right-click the 3D Perspective view and click Smooth Shaded from the pop-up menu).
Smoothing Group Clear All	This button removes the assigned Smoothing Groups from the Smoothing Group numbers. The polygons will remain smooth shaded, but they will no longer be in the same Smoothing Group.
Smoothing Group Auto Smooth wish	When assigning Smoothing Groups, ensure this is checked if you the selected polygons to be smooth shaded.

Team LRN

Table 13.13 Materials Tab Functionality

Item	Description
Material Selector Box	This is the white area at the top. It contains the names of the materials, one material per line. You always need to choose a material from this box before performing any material operations.
Material Preview	The currently chosen material is displayed, mapped onto a sphere. You can click and drag the sphere with the mouse to view hidden parts of the material map.
Ambient	Use this tool to get a color picker window for setting the ambient light of the environment the material is in. This attribute affects the color and the intensity of the color that the material reflects.
Diffuse	Use this tool to get a color picker window for setting the light that the material will directly reflect. This attribute has the most influence over the color of the material.
Specular & Specular Slider	Use this tool to set the specular highlight of the material. Basically, selecting a bright color will create a highlight on the material of the color chosen. Moving the slider below it changes the focus of the highlight. The highlight can range from appearing as a small spot to appearing as if the object is immersed in incandescent light.
Emissive	Use this tool to get a color picker window for setting the color and intensity of the light that the material emits. This attribute will appear as a glow around the material.
Transparency Slider	This slider adjusts the amount of transparency that an alphamap applies to a texture and the faces that the texture is assigned to. You must click Assign or click in the viewport to update the model to reflect your changes.
Texture Browse Button	Use this tool to select a texture to apply to the material. Pressing it will reveal a Windows Explorer browse box from which image files can be selected. The None button beside this button removes the texture file from the material.
Alphamap Browse Button	Use this tool to apply an alphamap to the material. A black-and-white image can be used to remove areas of texture where there may be holes. Black is fully occluded and white is fully visible; it is possible to use variations of gray to achieve semitransparency. The None button beside this button removes the alphamap file from the material.
New	The New button, when pressed, will create a new blank material with default attributes, no texture or alphamap files, and a default name.
Delete	To delete a material, choose it in the Material Selector, and then click the Delete button. This literally removes the material from the workspace, so use this wisely.
Rename Button & Box	To rename a material, choose it in the Material Selector box, type the desired name in the box next to the Rename button, and then click the Rename button.
Assign	Use this tool to assign the chosen material in the Material Selector box to the selected group.

Team LRN

The Joints Tab

Using the Joints tab (see Figure 13.21), you can specify the joints for skeletons, which are used in animations. Joints are also used as substitutes for the concepts of special nodes that are used by the Torque DTS Exporter. Table 13.14 describes the Joints tab functions.

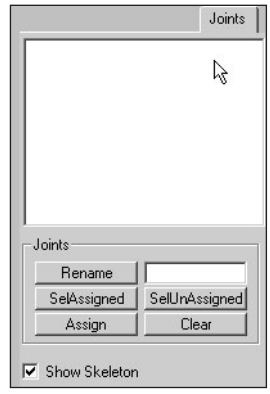


Figure 13.21 The Joints tab.

The Keyframer

The Keyframer (see Figure 13.22) is a special tool used for defining animations for your model. With it, you can save skeletal positions in a model. You then produce animation by storing several keyframes to the Keyframer and playing them back. There is a set of controls for managing the playback. Typically, only frames where changes take place need to be set by the user; hence the term *keyframe*—they are key to the animation. MilkShape 3D will fill in the pose or position frames between the keyframes. You must click the Anim button at the lower right in order to work with the Keyframer.

Table 13.15 describes the primary Keyframer functions.

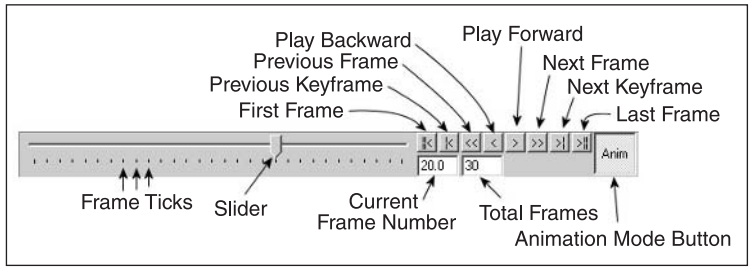


Figure 13.22 The Keyframer.

Table 13.14 Joints Tab Functionality

Item	Description
Joint Selector Box	This is the white area at the top. It contains the names of the joints, one joint per line. You always need to choose a joint from this box before performing any operations on joints.
Rename Button & Box	These tools work the same as the Rename tools in the other tabs: Choose a joint, type a new name in the box, and then click the Rename button.
SelAssigned	After you have chosen a joint, press this button to select all the vertices assigned to that particular joint.
SelUnAssigned	Press this button to select all vertices not assigned to the chosen joint.
Assign	Use this tool to assign vertices to a joint. To do this, choose the Select-Vertices tool from the Model tab, highlight the joint in the Joint Selector box that you wish to assign the vertices to, and then select the vertices and click Assign.
Clear	Press this button to clear all the assigned vertices from belonging to the chosen joint in the Joint Selector box.

Table 13.15 Keyframer Functionality

Component	Description
Keyframe Slider	Use the slider to preview your animation before playing it. Using the slider, you can move freely backward and forward between the frames with mouse movement instead of pressing the Play Forward and Play Backward buttons to see the animation. It is useful for selecting animation frames in smaller animations; use the Frame Position box to select frames for larger animations with many (more than a dozen or so) frames.
Playback Controls	The playback controls allow you to view your animation in MilkShape 3D in a manner similar to a VCR or DVD player. From the left, the buttons are First Frame, Previous Keyframe, Previous Frame, Play Backward, Play Forward, Next Frame, Next Keyframe, and Last Frame. All of these commands update the model to the current frame, and the slider is also moved to the appropriate frame.
Current Frame Number Box	Use this box when you have a lot of frames in your animation and the slider does not allow the accuracy you desire when selecting frames. You can type in a value here that will set the number of frames in the animation. The box will accept a whole number to indicate the frame to which you wish to go; the slider and view will change to reflect the selected frame.
Total Frames Box	In this box, enter the number of frames you want in your animation. Most modelers choose a relatively high number, depending on the number of animations the model is to perform, and key in animations between certain numbers of frames leaving a three- or four-frame gap between animations. With a Run, Walk, Jump, and Shoot animation, you would key in the Run animation first and then leave several frames, key in the Walk animation and then leave several frames, and so on.
Animate Button	This button enables the Keyframer. It behaves like a toggle—when down, the Keyframer is enabled; when up, the Keyframer is disabled.

The Preferences Dialog Box

The Preferences dialog box (see Figure 13.23), which you reach by choosing File, Preferences, has two tabs. The Viewport tab is used to set up the viewports' attributes, and the Misc tab offers miscellaneous settings. Table 13.16 provides details about each setting in the two tabs.

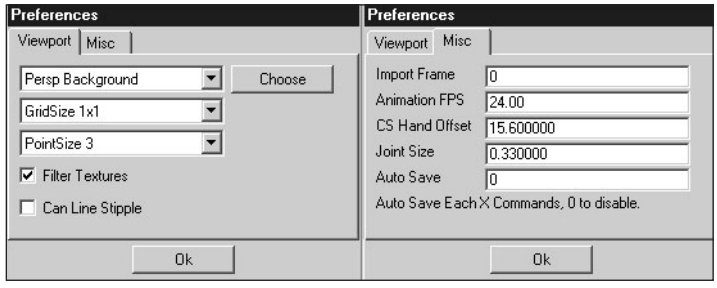


Figure 13.23 The Preferences dialog box.

Table 13.16 Preference Choices

Component	Description
Property Selector	With MilkShape, the user can customize the colors of components used when modeling. The drop-down list contains the component names; a color for the selected component can then be chosen by clicking the Choose button next to the drop-down box. Following is the complete list of color customizable components: perspective background (the 3D view) orthographic background (the 2D views) perspective grid orthographic grid X-Axis Y-Axis Z-Axis vertex selected vertex face selected face bone selected bone selected joint keyed bone
Grid Size	Use this control to set the spacing of the grid lines in the wire-frame views. The default grid size is 1×1; this gives the smallest line spacing. The grid size you use usually depends on the scale of the models you are building.
Point Size	Use this control to specify the size of the vertex points displayed in the wire-frame views. Larger point sizes are easier to see and to select individually, but they may tend to obscure model details in crowded areas at low view magnifications.
Filter Textures	When set, this turns on mipmapping texture filters. This will smooth the texture so that the rasterized pixels are not as noticeable.
Can Line Stipple	When moving, scaling, or extruding objects, MilkShape draws a guideline that indicates the vector of the action, denoting its direction and magnitude. This is usually a solid line, but with this option set, it is rendered as a dashed or dotted line. This also stipples the box line used for multiple selections.
Import Frame	This allows the user to specify the animation frame to be imported from MD2 or MD3 files using the Morph Target Animation mechanism.
Animation FPS	This specifies the playback speed of animations in *Frames Per Second* (FPS).
CS Hand Offset	This is used to specify the offset for either side of a decompiled CounterStrike model.

continued

Team LRN

Joint Size	This allows the user to set the display size of the joints that are used in MilkShape. You should change the size to reflect the scale of the model you work with.
Auto Save	This option allows you to specify how often the program will automatically save your work. The frequency is defined by how many commands or operations you want to be able to perform before the save happens. This option can be a lifesaver but can also be a nuisance if you set the value too low—especially if you are doing a lot of experimenting and undo your previous operations frequently. A setting of about 10 seems to work well.

Other Features

MilkShape has a few other features that we won't cover in great depth, but two that deserve at least an honorable mention are the Texture Coordinate Editor and the Message Panel.

The Texture Coordinate Editor provides primitive texture-mapping capability. It has some rather severe limitations that prevent it from being used in even moderately complex models. The biggest limitation is that it doesn't unwrap meshes independently. For this reason we use external tools, like UVMapper. UVMapper may be a bit more awkward to use, because it isn't integrated, but it does a better job, providing more flexibility and control.

The Message Panel displays output from executing plug-ins and modeling operations. It can be useful for providing insight into how MilkShape does its work, but its downfall is the screen space it takes up.

UVMapper

Earlier in this chapter, and even earlier than that in Chapter 9, we used the UVMapper program created by Steve Cox to help us skin a model. As promised, here is the section with the detailed information on UVMapper. We won't cover every detail. Instead, we will concentrate on those details that we can apply to our own needs here in this book.

The first thing to know about UVMapper is that it only operates on models saved in OBJ format, as created by the Alias Wavefront program. The UV unwrapping principles involved are the same for all similar tools. The author of UVMapper has also created UVMapper Pro, a newer release with many more features and greater flexibility. The companion CD includes a demo of UVMapper Pro, a restricted version (you can't save output, which, of course, we need to do). If you want to check out the enhanced features later, go ahead and poke around.

Team LRN

The File Menu

As is true in most programs, UVMapper's File menu provides commands for loading, saving, importing, and exporting files. See Table 13.17 for descriptions.

Table 13.17 UVMapper File Menu

Command	Description
Load Model	Load a Wavefront OBJ formatted model from file. After it is loaded, you will see the texture map layout in the UVMapper window. If you don't, then there are no texture coordinates included in the model. You can fix this by choosing Edit, New UV Map (see Table 13.18).
New Model	This command gives you a method for adding or creating your own models from shape primitives. The primitives are box, cone, cylinder, sphere, and torus.
Import UVs	With this command you can import UV coordinate data that has been saved separately from a model.
Save Model	Use this command to save the UV mapping data you've created along with the model you originally imported.
Save Texture Map	You can save the texture map image using this command. You can then load that image as a template into a program like Paint Shop Pro in order to apply that "artistic magic."
Export UVs	With this command you can export only the UV texture coordinates you've created using this program, without the rest of the model data.

The Edit Menu

The Edit menu is where the real power of UVMapper resides. Table 13.18 provides more information.

The Help Menu

The Help menu provides the user some assistance when working with the program. Table 13.19 provides more detail, and Table 13.20 provides a list of UVMapper hot keys.

UV Mapping

When you choose Edit, New UV Map you will be presented with a choice of five different unwrapping methods:

- Planar
- Box
- Cylindrical
- Cylindrical Cap
- Spherical

Table 13.18 UVMapper Edit Menu

Command	Description
Settings	Here you can specify how many pixels on your screen correspond to a single measurement unit. The value you use depends on the scale of the model you are working with.
Select By	This command gives you the ability to select on-screen objects by facet (face) or by selecting the vertices. Usually you leave this set to Facet.
Color	This command will let you indicate how you want to discriminate the different parts of the display. Your choices are Black and White (no discrimination), by Group, by Material, and by Region. This capability is handy when dealing with a complex model.
Tools	This command provides three different functions: Fix Seams, Split Vertices, and Weld Vertices. MilkShape offers these same abilities, but it's nice to know we have access to them here as well.
Select	With this command you can refine your object selection ability. There are five modes: All, None, by Group, by Material, and by Region. The by Group, by Material, and by Region options each provide a Selection dialog box if these entities actually exist in the model data. Judicious naming of groups (meshes) when in MilkShape can be a great boon when working here in UVMapper.
Assign	Use this command to assign selected objects to an existing group, material, or region. Again, you would normally do this in your modeling program, but it's nice to have the ability here if you realize you've forgotten to assign some faces to a particular group.
Rotate	This command allows you to rotate a selection around any of the three axes—or all three at once, if you want.
New UV Map	This command provides several different unwrapping methods: Planar, Box, Cylindrical, Cylindrical Cap, and Spherical. The options available here are quite extensive so they warrant coverage in their own section, called "UV Mapping," in this chapter.
Tile	This command is complementary to the Select command. Using Tile you can specify how the program displays the different parts of the model; they can be visually organized (tiled) according to group, material, or region.

Each of these methods is described in more detail here. Sometimes, even when you know exactly what the unwrapping method is supposed to do, you will be surprised at the results, so don't be afraid to experiment. Once you've loaded a model, you can keep trying the different unwrapping methods with different settings. Each time you do it, the program begins from scratch, so you don't have to worry about undoing your previous efforts.

Team LRN

Table 13.19 UVMapper Help Menu

Command	Description
Statistics	This will report the current status of your model. This will tell you the total vertices, textures, normals, facets, groups, and materials. Bear in mind that while you are editing a model, UVMapper will temporarily increase the number of texture coordinates allocated to the model, and so this is not a good representation of the actual number of texture coordinates the model will have upon saving. A more accurate way to obtain this information is from within the MilkShape modeling tool.
Dimensions	This will give you the overall geometric dimensions of the model. This will report the minimum and maximum values along each of the three axes (X, Y, and Z).
Hot Keys	This command will give you a list of the available hot keys. Table 13.20 also contains a list of the UVMapper hot keys.
About UVMapper	This command gives you information about the version, how to contact the author, and where you can obtain an updated version of the program.

Planar

When you use the Planar method, you will be presented with the dialog box depicted in Figure 13.24. Table 13.21 provides details about using the Planar method.

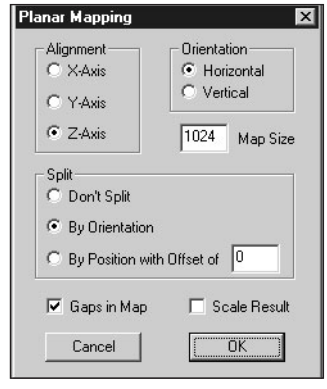


Figure 13.24 The Planar Mapping dialog box.

Box

When you use the Box method, you will be presented with the dialog box depicted in Figure 13.25. You can get more information on using the Box method in Table 13.22.

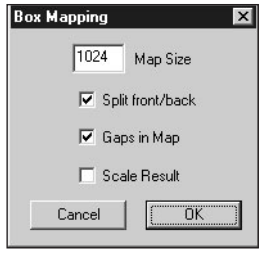


Figure 13.25 The Box Mapping dialog box.

Cylindrical

When you use the Cylindrical method, you will be presented with the dialog box depicted in Figure 13.26. Table 13.23 provides details about using the Cylindrical method.

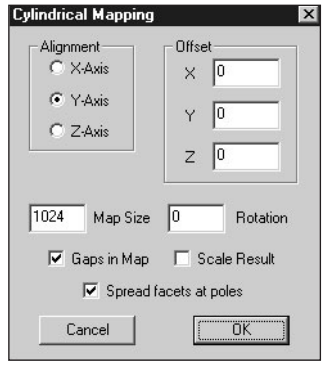


Figure 13.26 The Cylindrical Mapping dialog box.

Table 13.20 UVMapper Hot Keys

Key	Description
Esc	clears selection, undoes changes
Enter	clears selection, saves changes
Shift+number key	increases resize/movement amount
keypad *	quadruples size of selection
keypad /	quarters size of selection
keypad +	increases size of selection
keypad -	decreases size of selection
keypad #	moves selection
=	maximizes selection
.	snaps selection to facets
[	hides selected facets
]	shows selected facets
\	toggles facets on and off
'	hides unselected facets
uU/vV	resizes selection (fine)
x/X/y/Y	resizes selection (coarse)
Ctrl+x	inverts selection horizontally
Ctrl+y	inverts selection vertically
Ctrl+b	loads background
Ctrl+c	clears background
Ctrl+u	flips background horizontally
Ctrl+v	flips background vertically
Tab	toggles background display
t	triangulates object
Insert	checks for degenerate facets

Cylindrical Cap

When you use the Cylindrical Cap method, you will be presented with the dialog box depicted in Figure 13.27. Table 13.24 provides details about using the Cylindrical Cap method. This method is similar to the Cylindrical method, except that it assumes you are unwrapping a cylinder with end caps, as if there were closed lids on both ends of a can. The caps are mapped separately from the tubing of the cylinder.

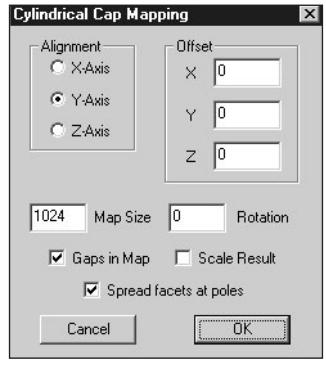


Figure 13.27 The Cylindrical Cap Mapping dialog box.

Table 13.21 Planar Mapping Options

Option	Description
Alignment	This allows you to specify the axis along which the model will be mapped.
Orientation	This allows you to alter the layout of the texture map template. It only has an effect when using the Split option (described later in this table). If you select Don't Split, the Orientation option has no effect. When splitting the model into front and back sections, you can have the two halves side by side (Horizontal) or above and below each other (Vertical). Which you want to use really depends on the geometry of the model. If you don't like the layout of the texture map after using planar mapping, try changing this option.
Map Size	This will specify the maximum dimension of the texture map template. Depending on the model it may be vertical or horizontal, but the texture map is guaranteed not to exceed this value in either width or height. One side will equal this value, and the other will be scaled accordingly.
Split	This option allows you to divide the texture map into front and back sections. (To adjust the placement of these sections, see the Orientation option earlier in this table.) You have three options. Don't Split will give you one map with the front and back facets on top of each other. By Orientation will calculate the facet normals, placing all facets that face toward the eye on one side, and placing all facets that face away on the other. By Position with Offset of allows you to divide the model based on geometry rather than facing. Using an offset of 0 will divide the model in half. You can adjust this offset to change how many facets are on each side.
Gaps in Map	This allows you to separate the sides of the box on the texture map. If the sides touch, sometimes you will see one pixel of the side on the front, for example.
Scale Result	Use this option to specify how much larger or smaller the resulting texture map should be.

Table 13.22 Box Mapping Options

Option	Description
Map Size	This will specify the maximum dimension of the texture map template. Depending on the model it may be vertical or horizontal, but the texture map is guaranteed not to exceed this value in either width or height. One side will equal this value, and the other will be scaled accordingly.
Split front/back	Setting this option will divide the model into six sections: front, back, top, bottom, left side, and right side. Uncheck this option if you want to combine top and bottom, left and right, front and back, giving you only three sections.
Gaps in Map	This allows you to separate the sides of the box on the texture map. If the sides touch, sometimes you will see one pixel of the side on the front, for example.
Scale Result	Use this option to specify how much larger or smaller the resulting texture map should be.

Team LRN

Table 13.23 Cylindrical Mapping Options

Option	Description
Map Size	This will specify the maximum dimension of the texture map template. Depending on the model it may be vertical or horizontal, but the texture map is guaranteed not to exceed this value in either width or height. One side will equal this value, and the other will be scaled accordingly.
Split front/back	Setting this option will divide the model into six sections: front, back, top, bottom, left side, and right side. Uncheck this option if you want to combine top and bottom, left and right, front and back, giving you only three sections.
Gaps in Map	This allows you to separate the sides of the box on the texture map. If the sides touch, sometimes you will see one pixel of the side on the front, for example.
Scale Result	Use this option to specify how much larger or smaller the resulting texture map should be.

Table 13.24 Cylindrical Cap Mapping Options

Option	Description
Alignment	This allows you to specify the axis around which the model will be mapped.
Offset	When mapping a model with one of these methods (Cylindrical, Cylindrical Cap, or Spherical) the model is mapped around a center point. This center is calculated using the maximum and minimum geometry values along each axis. This works quite well for mapping a true sphere or cylinder, but if you have a model that is, say, a sphere with a spike on the side of it, the calculated center may not be what you want. To adjust the center of the model from what's been calculated, use this option.
Map Size	This will specify the maximum dimension of the texture map template. Depending on the model it may be vertical or horizontal, but the texture map is guaranteed not to exceed this value in either width or height. One side will equal this value, and the other will be scaled accordingly.
Rotation	Use this to specify how much, if any, rotation will be applied to the resulting texture map image template.
Gaps in Map	This allows you to separate the sides of the box on the texture map. If the sides touch, sometimes you will see one pixel of the side on the front, for example.
Scale Result	Use this option to specify how much larger or smaller the resulting texture map should be.
Spread facets at poles	Oftentimes facets are squeezed together when the mapping occurs, especially at places like the "poles" (the tops and bottoms of the map, just like on maps of the Earth). With this option set, the resulting map will spread the facets at the poles to alleviate the pinching effect.

Team LRN

Spherical

When you use the Spherical method, you will be presented with the dialog box depicted in Figure 13.28. You can get more information on using the Spherical method in Table 13.25.

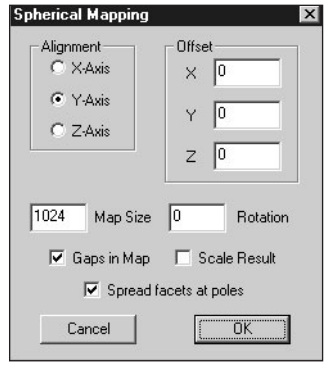


Figure 13.28 The Spherical Mapping dialog box.

Table 13.25 Spherical Mapping Options

Option	Description
Alignment	This allows you to specify the axis around which the model will be mapped.
Offset	When mapping a model with one of these modes (Cylindrical, Cylindrical Cap, or Spherical) the model is mapped around a center point. This center is calculated using the maximum and minimum geometry values along each axis. This works quite well for mapping a true sphere or cylinder, but if you have a model that is, say, a sphere with a spike on the side of it, the calculated center may not be what you want. To adjust the center of the model from what's been calculated, use this option.
Map Size	This will specify the maximum dimension of the texture map template. Depending on the model it may be vertical or horizontal, but the texture map is guaranteed not to exceed this value in either width or height. One side will equal this value, and the other will be scaled accordingly.
Rotation	Use this to specify how much, if any, rotation will be applied to the resulting texture map image template.
Gaps in Map	This allows you to separate the sides of the box on the texture map. If the sides touch, sometimes you will see one pixel of the side on the front, for example.
Scale Result	Use this option to specify how much larger or smaller the resulting texture map should be.
Spread facets at poles	Oftentimes facets are squeezed together when the mapping occurs, especially at places like the "poles" (the tops and bottoms of the map, just like on maps of the Earth). With this option set, the resulting map will spread the facets at the poles to alleviate the pinching effect.

Moving Right Along

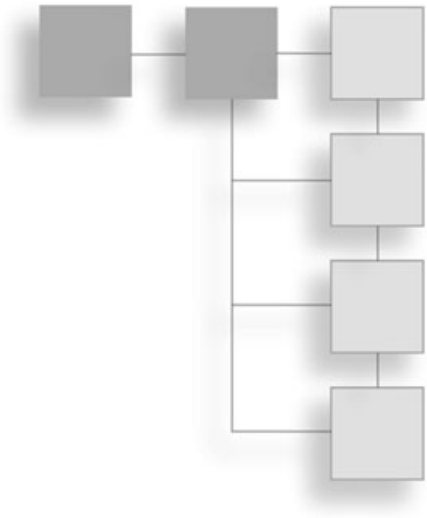
Well, there you have two pretty comprehensive, low-cost modeling tools: MilkShape 3D by Mete Ciragan and UVMapper by Steve Cox. These guys have done an admirable job creating these programs in the shareware or freeware spirit. They deserve not only a round of applause and big thank you, but you could also perhaps send a few dollars their way by registering their shareware programs. The cost is minuscule, and the benefits are great.

By using the common Wavefront file format, we can use each tool in complementary ways to create models for our games. This is a pretty common theme; notice also that we've used Paint Shop Pro—another low-cost tool—in the same way in conjunction with MilkShape. In the next few chapters we will tackle the Big Jobs: animated characters, vehicles, and weapons. It would not be wasted effort if you wanted to take some time out at this point to practice by designing and building some models to your own specifications.

The more you use a tool, make mistakes, and figure out what you did wrong and then make any necessary corrections on your own, the more proficient you will become.

Team LRN

CHAPTER 14



Making a Character Model

In this chapter we are going to build a character model, step by step. We are going to animate it and skin it. It's going to be a long and sometimes hectic ride, so hang on to your hat!

Modeling Techniques

There are many different approaches or techniques that modelers use. The differences can be based on what tools are available to do a given job or what data is available about the item to be modeled. There are other techniques available not described here—that's because we are modeling for games, and low-poly modeling is the philosophy we need to follow. Remember that the more polygons one model uses, the fewer polygons available for other instances of that model, or for other models, in a given rendered frame of a scene at a given frame rate. In games there is that polygon budget to consider.

Shape Primitives

Creating models using shape primitives can be an extremely quick way to build a low-poly model, depending on your expertise and eye for detail. The basic technique involves selecting a primitive that best matches the part of the model you are building. The primitive shape must contain enough polygons and vertices for you to adjust the shape to closely match your target.

This is the technique we are going to use to build our character model in this chapter.

Incremental Polygon Construction

Incremental Polygon Construction is a method that fairly closely approximates modeling with clay in the real world. Sculpting with clay generally involves adding bits of clay

Team LRN

together to create shapes that grow in size and detail. The clay can be poked and prodded, smoothed and pinched until it accurately depicts the item being modeled.

With Incremental Polygon Construction, the process is similar: We apply vertices in 3D space that represent high points and low points of the features to be modeled, and then we build triangles or faces connecting these vertices. One point of departure from clay modeling is that we typically don't add faces on top of (such that they completely obscure) other faces, because we have no need for a solid to give us the required volume. But the principle of adding faces to the existing topology of a model as the model grows is the same, as is the useful concept that we add only what we need, and no more.

The best way to get started with modeling in this way is to use photographs or sketches of the target from several directions: from directly in front, directly above, and one or both sides. From the pictures we can obtain the locations and shapes of the features and their high and low points. We mark these points in our 3D views, and then we proceed to build faces from them.

This technique can be quite slow going. It is also prone to errors that are difficult to correct, because you may have moved dozens of steps beyond where the real error occurred before the error becomes evident.

Axial Extrusion

In the simplest sense, you start with a primitive object (usually a box, but it could be a simple facet or triangle), subdivide it, and then select specific polygons to extrude into meshes to form general shapes. When you subdivide objects, you increase the number of polygons on each side of the shape. You then adjust and refine the extruded polygons to form the details of the model. This approach is similar to making models of geographic terrain using a contour map as a guide, with cardboard or plywood sheets to build up the terrain, and then smoothing the edges with some kind of filler.

With Axial Extrusion, you limit your extrusions to one of the three axes— sometimes all three in various combinations—but individual extrusion only occurs in one axis. This technique is usually restricted to inanimate objects, but sometimes certain parts of character models are made this way.

One example of using axial intrusion in character modeling is when creating a head. A series of flat-plane profiles (called *cuts*) are made of a head, after which each profile is extruded once in each direction on the transverse axis (the axis that runs from ear to ear). Then each extruded mesh is married to the extruded meshes of the adjacent cuts by an averaging of the vertices. You'll actually get to do some of these extrusions later in this chapter, and others.

Arbitrary Extrusion

Arbitrary Extrusion has much in common with Axial Extrusion, except that you extrude your base primitive shapes in whatever directions are necessary. Like Incremental Polygon Construction, this approach to modeling can be seen as similar to sculpting in clay. Machinery lends itself well to modeling with this technique.

Topographical Shape Mapping

Topographical Shape Mapping is a method usually used to model terrain, like Axial Extrusion often is, except that Topographical Shape Mapping is best suited for automated operations rather than manually modeling.

In the geographic sense, topographic data can be obtained from various government and private sources. The data consists of, at a minimum, a coordinate and an altitude for each mapped point on the real terrain's surface. There are various algorithms and many programs available that can read this data from a file and render a 3D view of the terrain in question. The data files come in various formats depending on the agency that produces them: DLG-O, DEM, SDTS, and DRG, to name just a few from that acronymic world. Normally this approach is used in one of the many available *Geographic Information Systems* (GIS), and there are tools that can convert this data into a format you can use for modeling in games.

Hybrids

Well, the Hybrid category is the catchall category. Often it is prudent to combine techniques in a single model—use the approach that works best for the component being created. If you find yourself mixing techniques, most likely you will be doing a little bit of Incremental Polygon Construction mixed with many shape primitives or using a few primitives mixed with a great deal of Arbitrary Extrusion.

The best point to be made here is that you should use what works best for you in your current circumstances.

The Base Hero Model

The technique we are going to use is basically the Shape Primitives approach. We will hand-modify various shape primitives to get the results we want.

The kind of model we are going to make is primarily a *segmented-mesh model*. An alternative would be a *continuous-mesh* model. The difference is that in the segmented-mesh model, there are different, distinct objects or meshes for different components in the model, whereas in the continuous-mesh model, the entire model has one large, convoluted surface. Our primary segments will be as follows:

Team LRN

- head
- torso
- right leg
- left leg
- right arm
- left arm

So that's six segments in all. (A continuous-mesh model would have one segment.) All the leg and arm segments will each have two subsegments. Each segment or subsegment can be thought of as an individual mesh, or submesh.

The Head

We'll use the Shape Primitives approach to build the head. The keys to successful use of this technique are (1) choose the right primitive and (2) use a primitive with sufficient vertices to do the job.

For the head part of the model, we're going to use a cylinder with 12 faces on the tube, stacked 6 segments high. That translates to a 6-stack, 12-slice cylinder, in MilkShape terms.

1. Open MilkShape, create a new document, and set the Point Size to 3 and the Grid to 1×1 in the Preferences dialog box. Save the new file as C:\3DGPAi1\resources\ch14\myhead.ms3d.
2. Create a 6-stack, 12-slice cylinder as depicted in Figure 14.1. Size the cylinder such that the bounds of the cylinder extend from about −20 to +20 on all three of the axes.
3. Choose Select in Vertex mode and then select the bottom layer of vertices.
4. Scale the selected bottom vertices to 95 percent of original, as depicted in Figure 14.2.

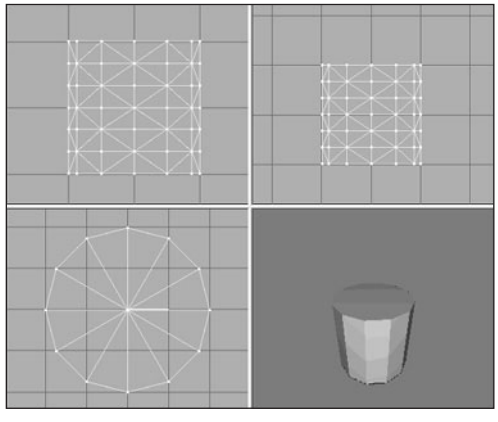


Figure 14.1 The initial cylinder.

5. Now select the top five rows of vertices, ignoring the bottom two rows, and scale them to 95 percent.
6. Next, scale the top four rows of vertices to 95 percent.
7. Repeat the scaling operation for the top three, then the top two, and finally the top row by itself. You should now have a cylinder with a bit of a bevel at the bottom that tapers gently toward the top, as shown in Figure 14.3.

8. Next, shift the top five layers of the cylinder toward the back, so that the rearmost vertices (designated A, highlighted in black, in Figure 14.4) line up, at the back, with the layer of vertices that is second from the bottom (B in Figure 14.4). They don't have to be aligned precisely, but try to get them pretty close, as shown in Figure 14.4.
9. Next, working from the Side view (Top Right viewport), select the bottom six vertices visible in that view (at the right side of the view) and move them down and to the right a bit. Figure 14.5 shows which vertices you want and how far to move them. These vertices make up the jaw.
10. Select all the vertices in the model, and scale to 75 percent in the Y-axis only. Do this by typing the value **0.75** into the Y scale box when you have the Scale tool selected and then clicking Scale.

tip

The view in what MilkShape calls the Left viewport is for us actually the Right view (or Right Side view) located in the upper-right frame, because Torque's coordinate system is oriented differently. It's because of this that I normally use MilkShape with the Show Viewport Caption option under the Window menu turned *off*, in order to avoid confusing myself.

11. Now, using the same technique of selecting and moving (without doing any scaling) as used in steps 4 to 9 above, shape the model as near as you can get to Figure 14.6. This is the Right Side view (upper right frame). You only need to work in this view,

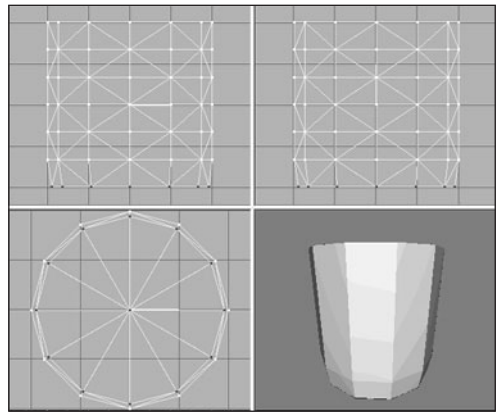


Figure 14.2 Selecting the bottom vertices.

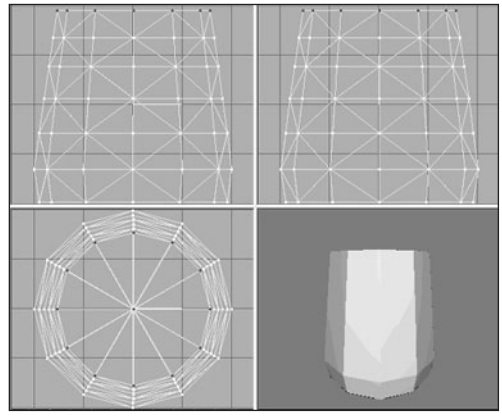


Figure 14.3 Tapering the cylinder.

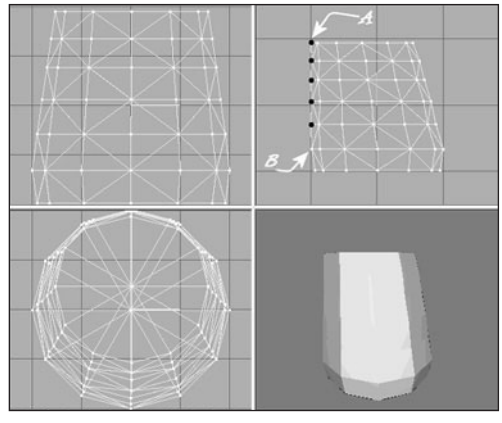


Figure 14.4 Shifting the layers.

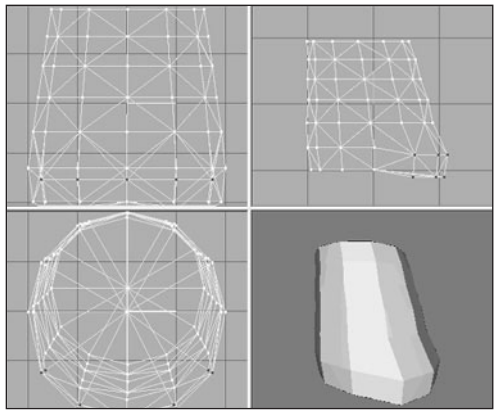


Figure 14.5 Shaping the jaw.

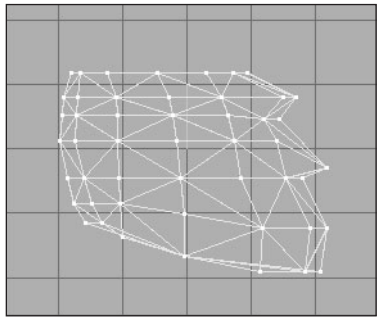


Figure 14.6 Shaping the head.

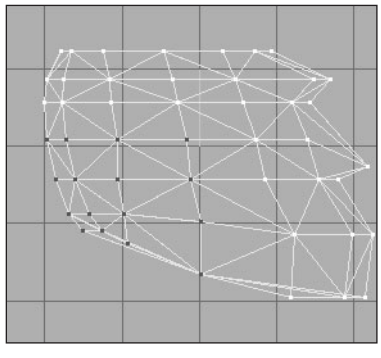


Figure 14.7 Back of the head/upper neck.

and no other, and only use the Select and Move tools. Now you can see the head shape taking form in profile, with the nose jutting out.

12. Okay, this next part gets a bit tricky. Using the Right Side view, select the 16 vertices in the lower-left corner (which is the lower back of the head/upper rear neck area), as shown in Figure 14.7.
13. Scale this group of vertices to 80 percent by typing **0.8** in the X-axis scale box, and then click Scale.
14. Now select just the nine vertices in the lower left, as shown in Figure 14.8, and scale these to 80 percent again.

    What this does is make the jaw and cranium parts of the head stand out in an exaggerated fashion. By doing the scaling incrementally on the vertices in the region like that, we get a fairly smooth shape. Take a moment to swivel the model around in the 3D view, and you can now see a definite cartoonlike big-jawed, low-browed heroic figure taking shape. Okay, so not *all* heroes look like that. But we're making a game, right? So make it fun!

    Now, as cute and lovable as that beetle-browed look is, it's a bit too Cro-Magnon and robotic looking, so we need to tone down the forehead and eyebrow area somewhat.
15. In the Right Side view, in the row of vertices that is second from the top (see Figure 14.9), select the vertex that is the second from the right (in the temple area) by dragging the Selection tool around it. This will have the effect of selecting that vertex and any others that are obscured behind it. There happens to be one more back there, so you will end up with two vertices selected, which you can see by examining the model in other views.

Team LRN

16. Drag the vertices back (to the left) a few ticks.
17. Switching now to the Front view (upper left frame), scale those two vertices by 120 percent in the X-axis. This has the effect of widening the gap between them. (See Figure 14.10.) These steps have the effect of softening the sharp corners, just enough to make the head more organic looking.

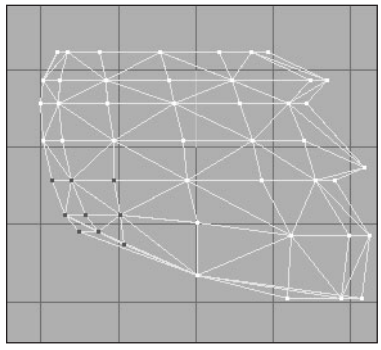


Figure 14.8 The smaller back of the head area.

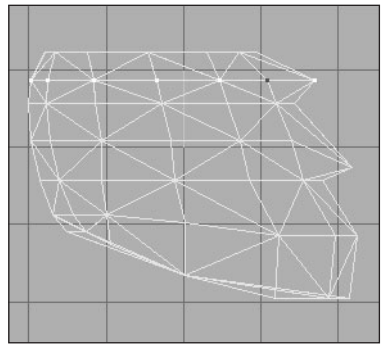


Figure 14.9 The temple vertices.

18. Still with the Front view, select all the vertices in the top three rows, which is mostly the cranium area, and then incrementally apply 90 percent X-axis *and* Z-axis scaling to them—as you did earlier: top three, then top two, and so on. Figure 14.11 shows the results we are looking for here.

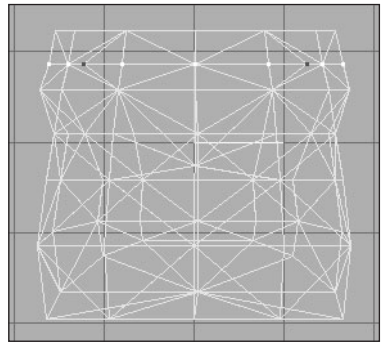


Figure 14.10 Scaling the temple vertices.

19. If you haven't saved your work recently, do it now. No particular reason, other than it's good practice. We're getting close to finished with the head.
20. Using Figure 14.12 as a guide, select the three ear vertices in the Right Side view.

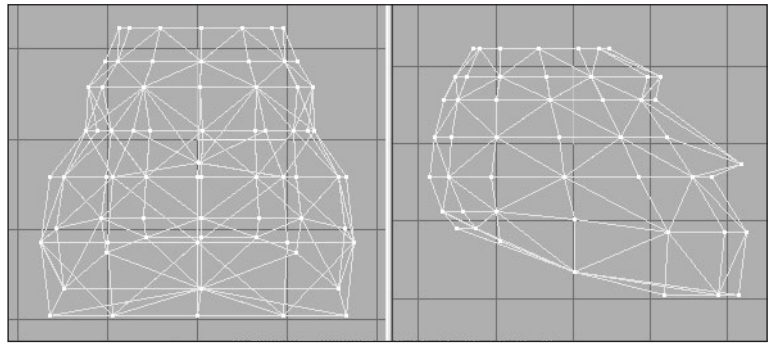


Figure 14.11 Scaling the cranium.

21. Stretch the ear vertices apart by scaling them 170 percent in the X-axis, as shown in Figure 14.13.

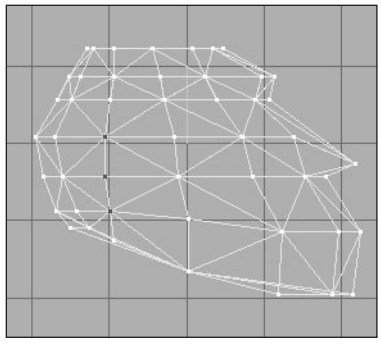


Figure 14.12 The ear vertices.

22. Now back in the Right Side view, guided by Figure 14.14, select the three columns of vertices at the rear of the head.
23. Drag them forward so that the rightmost column of selected vertices is just behind the unselected column (the fifth column), as shown in Figure 14.15.
24. Next drag the two columns at the back of the head forward, so that you end up with a configuration like the one depicted in Figure 14.16.

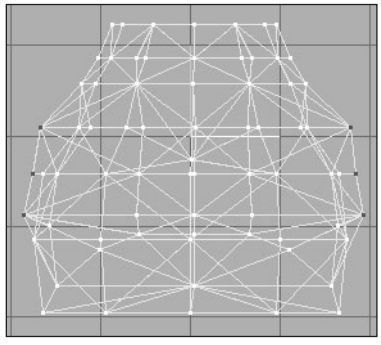


Figure 14.13 The scaled ear vertices.

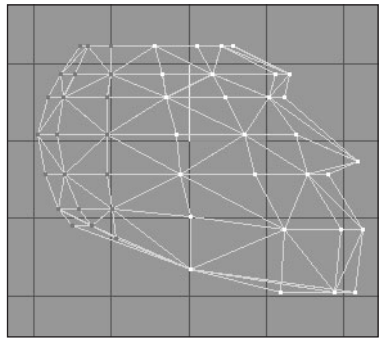


Figure 14.14 Selecting the three columns of vertices.

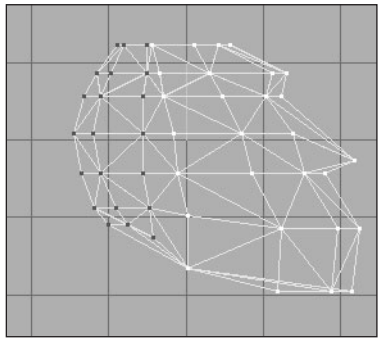


Figure 14.15 Dragging the vertices forward.

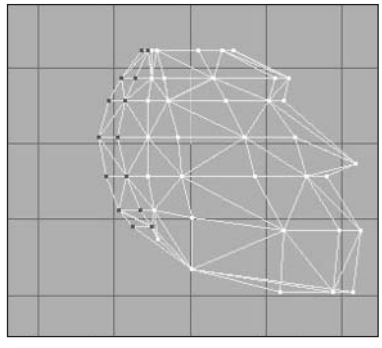


Figure 14.16 After dragging the vertices.

25. By now, you should be getting fairly adept at using the Select, Move, and Scale tools in MilkShape, so I'll give you a little assignment: Make the scalp region at the top of the head look like the scalp shown in Figure 14.17, using just these three tools and operating only on the top row of vertices. You will have to work in both the Front and Side views while monitoring your progress in the 3D view.

Team LRN

26. Next, use the same techniques to shape the nose and eyes. Figure 14.18 shows which vertices to use to shape the nose. Scale the vertices by 50 percent in the X-axis.
27. Shape the eye-socket vertices shown in Figure 14.19 by scaling to 30 percent in the X-axis.
28. Now, this entire work should exist as one group. Rename that group as "head" in the Groups tab in the toolbox.
29. Save your work as C:\3DGPAi1\resources\ch14\myhead.ms3d. By saving the head in its own file, you can keep it safely out of the way while you work on the other parts.

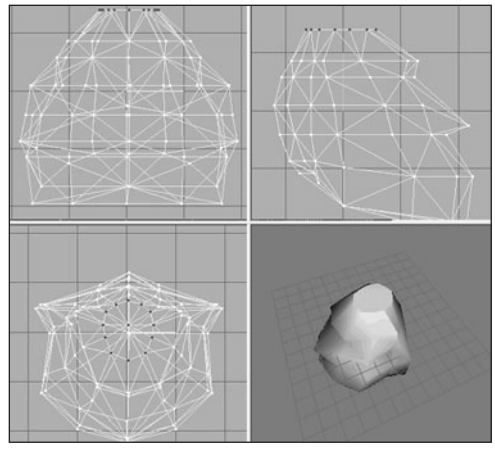


Figure 14.17 Shaping the scalp.

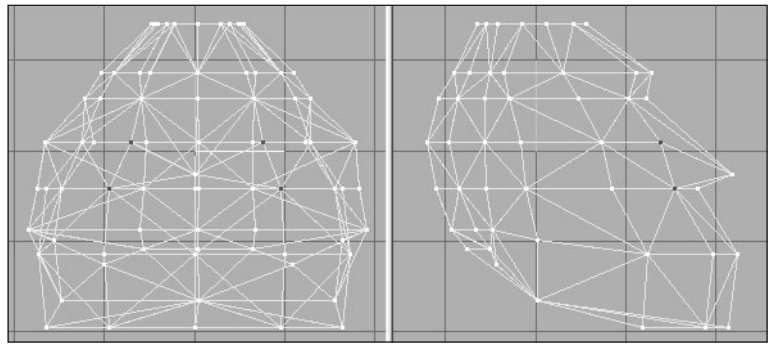


Figure 14.18 The nose vertices before scaling.

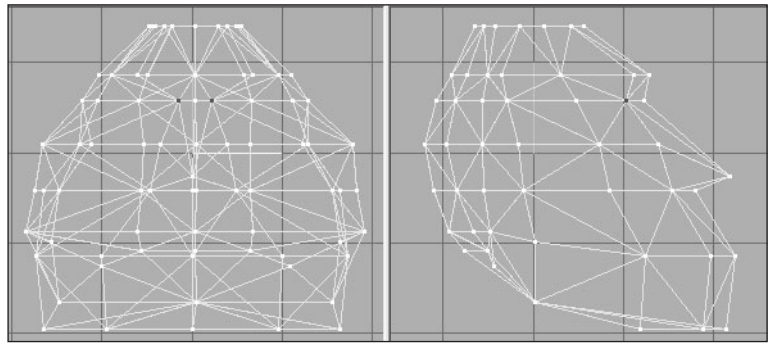


Figure 14.19 The eye-socket vertices after scaling.

And there you have it—as you can see in Figure 14.20, steely-eyed, big-jawed, beetle-browed genuine dyed-in-the-wool hero material!

The Torso

Like the head, the torso will be based on the cylinder shape, but this time we will use two of them and weld them together.

1. If you have the head file still open, leave it open. If you don't have it open, then open it.

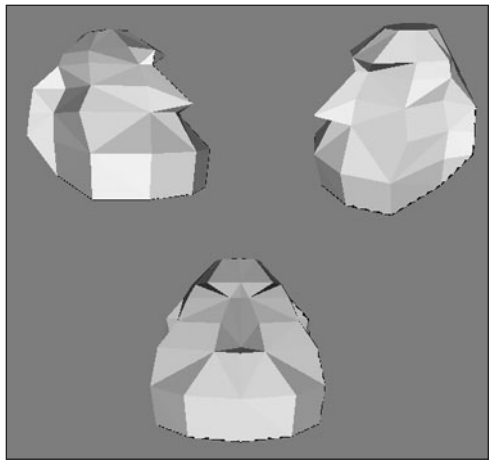


Figure 14.20 The finished hero head.

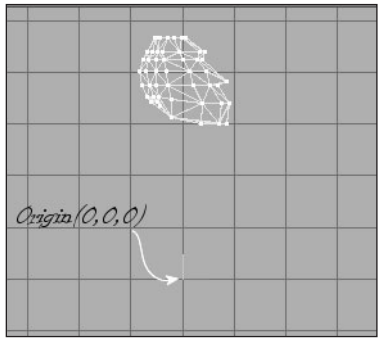


Figure 14.21 Positioning the head mesh.

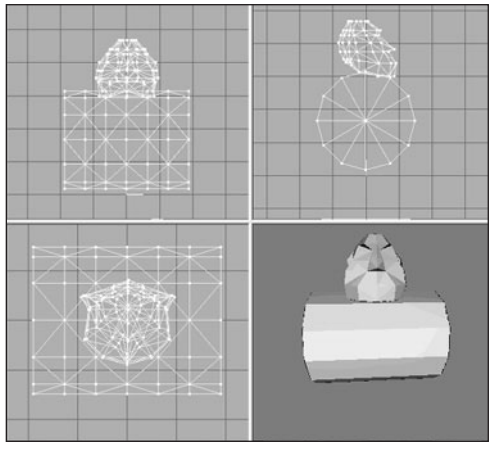


Figure 14.22 The relationship of the chest cylinder to the head.

2. Save the file as C:\3DGPAi1\resources\ch14\mytorso.ms3d. We want to have the head around to use as a sizing guide when we start the torso model, and then we will delete it.
3. Drag the head mesh up until it is three or four grid lines above the model origin (the 0,0,0 coordinate), as suggested in Figure 14.21.
4. Use the Cylinder shape, and make one that has 6 segments, or stacks, and 12 slices, or faces. Give the name "chest" to the group it creates.
5. Rotate the cylinder by 90 degrees in both the X- and Y-axes.
6. Move and scale the cylinder until it has the same relationship to the head, as shown in Figure 14.22.
7. Turn the Auto Tool option off, if it is on.
8. In the Front view, select all the vertices from one end of the cylinder, then hold down the Shift key, and drag over the vertices at the other end of the cylinder to select them as well. These vertices form the cylinder caps for either end.
9. Scale the vertices to 50 percent in the Y- and Z-axes.
10. Drag the vertices up until the top ones are in line with the top of the cylinder. Figure 14.23 shows what the result should look like.
11. If you like to use the Auto Tool option, turn it back on now.
12. In the Front view, select the right-hand end cap, and rotate it by −20 degrees in the Z-axis.
13. Now rotate the left-hand end cap by +20 degrees in the Z-axis.

14. In the Groups tab in the toolbox, choose the head group and delete it. This gets it out of the way so it won't clutter our model. We have the head saved separately, so no worries here.

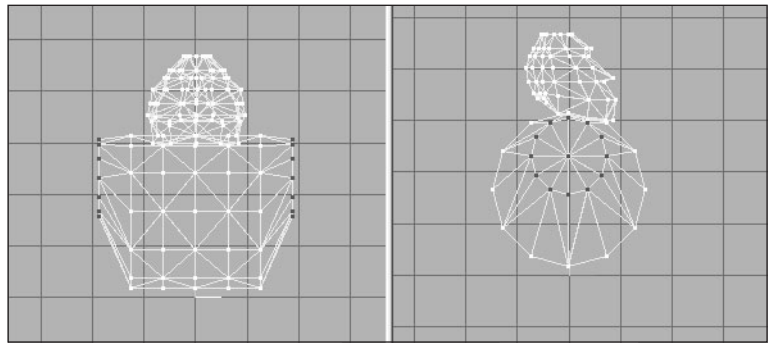


Figure 14.23 The cylinder caps after scaling and moving.

15. In the Top view, select the two vertices in the middle at the bottom, in the area of the sternum, as shown in 14.24, and move them toward the inside of the chest a bit. Use Figure 14.24 as a guide.
16. Now you'll do the same for the back as for the front, but just slightly differently, for a different effect. In the Front view, select all the vertices in the top three rows, including the ones that are in the end caps.
17. Hide these vertices, using Edit, Hide Selection.

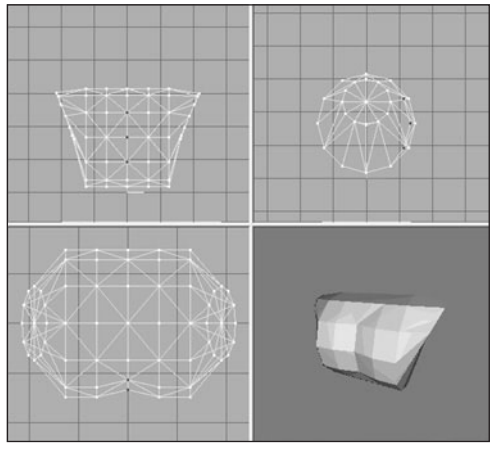


Figure 14.24 The sternum vertices after moving.

18. Now in the Top view, select the middle three vertices at the top of the view, as shown in Figure 14.25. These are the middle back vertices.
19. Move the middle back vertices toward the inside of the chest a bit, just as you did with the sternum, but perhaps not quite as much.
20. Create another new cylinder (to be named "ab"), and give it the same 90 degree rotation in the X- and Y-axes.
21. Move and scale the ab cylinder until it has the same relationship with the chest, as shown in Figure 14.26.

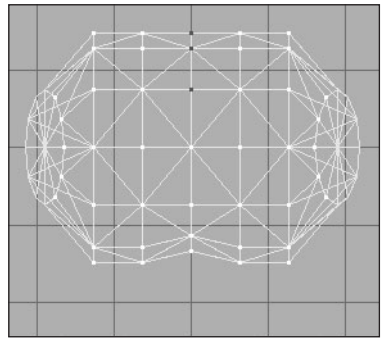


Figure 14.25 The middle back vertices.

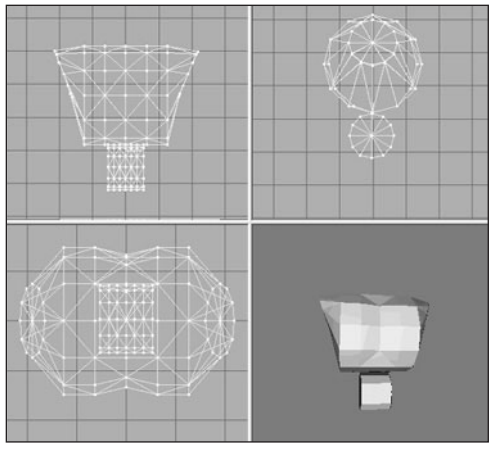


Figure 14.26 The ab cylinder relative to the chest.

Now we have our primitive abdomen inserted. We're going to have to splice that mesh onto the chest mesh in order to complete the torso. It's actually not terribly hard to do, and after you've done it once, it will seem intuitively easy. But there are quite a few fiddly little steps involved to get there from here. So please be patient.

22. Using the Groups tab, hide the ab mesh.
23. In the Right Side view, select the bottom vertices, as shown in Figure 14.27, and then hide them using Edit, Hide Selections.

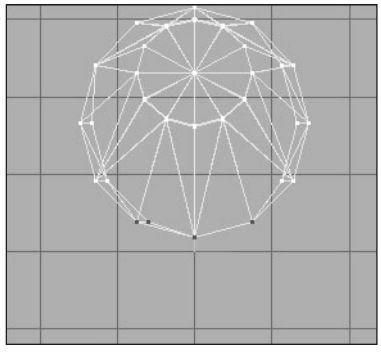


Figure 14.27 Hiding the lower chest vertices.

24. Back to the Groups tab, unhide the ab mesh. Don't use the general Unhide All command, because we want the chest vertices that we just hid to stay hidden.
25. In the Right Side view again, select the vertices shown in 14.28 and drag them up so they are directly over the location where the hidden vertices for the chest are. Study Figure 14.28, which shows the vertices selected and dragged into position. Compare it with Figure 14.27 to get a sense of the right place to put the vertices. The intersection of lines shown by the white arrow in Figure 14.28 does not get a vertex at this time—we will deal with that shortly.
26. In the Front view, locate the end cap vertices, as shown in Figure 14.29, and drag them out to the position indicated in that figure.
27. Next, do the same for the vertices to the left of the previous set. Drag them to exactly the same place as the previous set, as shown in Figure 14.30.
28. Repeat steps 26 and 27 for the other end of the ab mesh.
29. Drag the next set of vertices over to the chest positions, as shown in Figure 14.31.

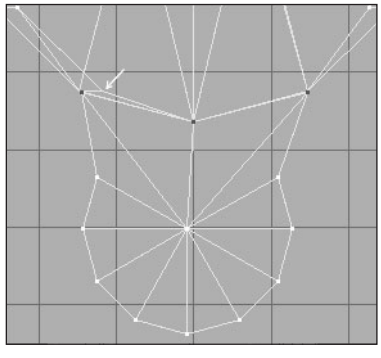


Figure 14.28 The ab vertices dragged over on top of the chest vertices.

30. Repeat the drag operation for the other end. You should now have something that closely resembles the layout in Figure 14.32.
31. Zoom in on all the places that you dragged vertices to, and make sure that they are exactly over the line intersections of the chest triangles.
32. In the Right Side view, select and hide all vertices on a line from the center of the cylinder forward (with forward being toward the right of the view). Figure 14.33 shows the vertices we're interested in.
33. Back in the Front view, select the center vertex at the top of the ab cylinder, as depicted in Figure 14.34. If you've done step 32 correctly, then as you scan around the other views, you will see that only one vertex has been selected.
34. Switch to the Right Side view and drag that lone vertex up to the spot that I pointed out with the white arrow way back when in Figure 14.28.

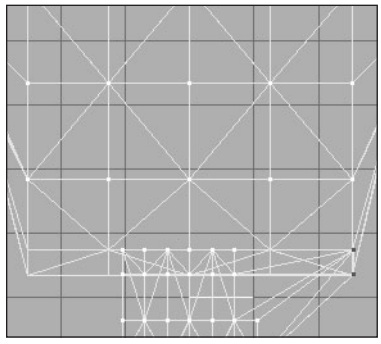


Figure 14.29 Dragging some end cap vertices over on top of chest vertices.

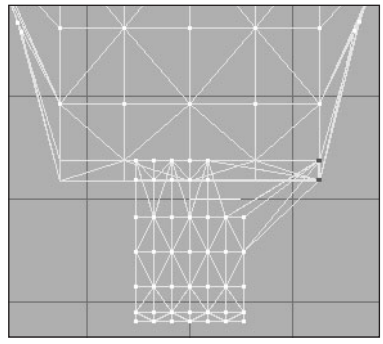


Figure 14.30 Dragging the end cap neighbor vertices over on top of the chest vertices.

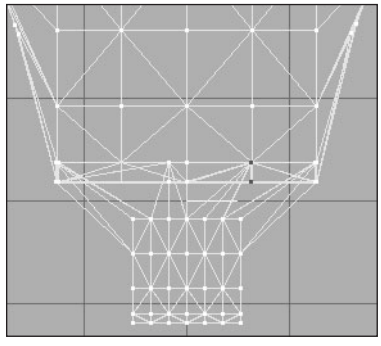


Figure 14.31 Dragging the next set of vertices into position.

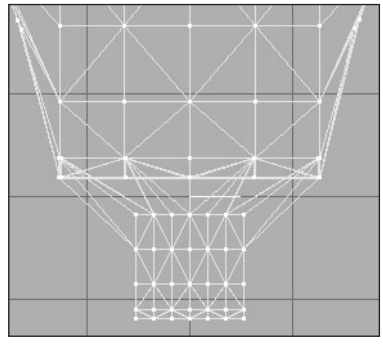


Figure 14.32 The final Front view layout.

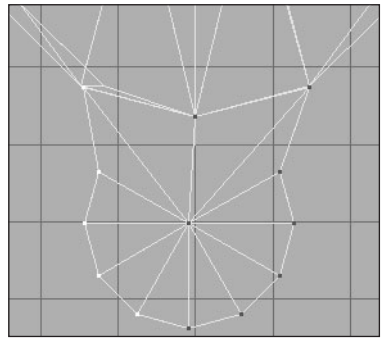


Figure 14.33 Select and hide these vertices.

You should now have a configuration that looks like the one shown in Figure 14.35. Again, take the time to zoom in and ensure that all the dragged vertices are exactly over the line intersections of the chest triangles.

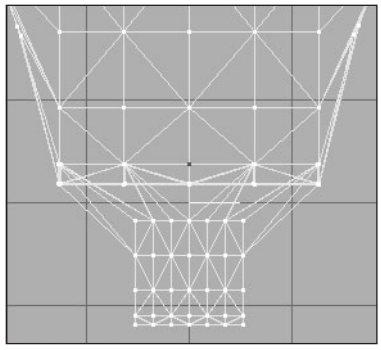


Figure 14.34 The top center cylinder vertex.

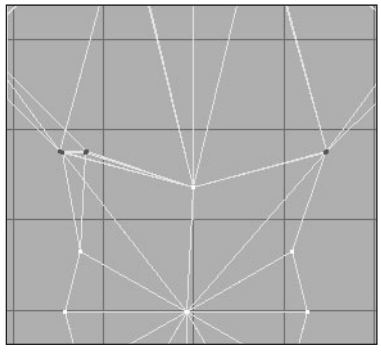


Figure 14.35 Selecting the common chest and ab vertices.

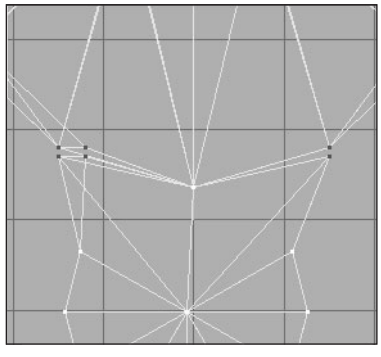


Figure 14.36 After snapping to grid…oops!

35. Unhide all the hidden vertices using the Edit, Unhide All command.
36. Select all the vertices from both meshes located at the places where you placed the dragged vertices. It's probably best to do this with the Right Side view, as I did in Figure 14.35. These are the vertices of each mesh, chest and ab, that share the same locations.
37. Choose Vertex, Snap To Grid. This should have the effect of forcing the closely adjacent vertices of each mesh to exactly align on the grid locations. However, if your vertices weren't aligned closely enough earlier, then they might diverge, as you can see happened to me in Figure 14.36. That's because I didn't take my own advice to zoom in and tweak each moved vertex position to be exactly right.

    It should be pretty obvious where the misaligned vertices have to go. If you have any that wandered off like I did, go back to the Right Side view and the Front view and move the wayward vertices into position. Then repeat the Snap To Grid operation.
38. Compare your results with the images in Figure 14.37, making sure you have the same thing as shown there.
39. If they aren't already selected, reselect the vertices shown back there in Figure 14.35.
40. And now the moment we've all been waiting for: Choose Vertex, Weld Vertices.

    All vertices that share identical common coordinates will be "welded" together. This basically means that superfluous copies of vertices will be deleted, and the polygons that we're defining will be reattached to the remaining single copy of each vertex.
41. In the Groups tab in the toolbox, choose both meshes, the chest and ab, so that they are both selected and highlighted in the wire-frame views.
42. Click Regroup, and then rename the newly consolidated group as "torso".

Team LRN

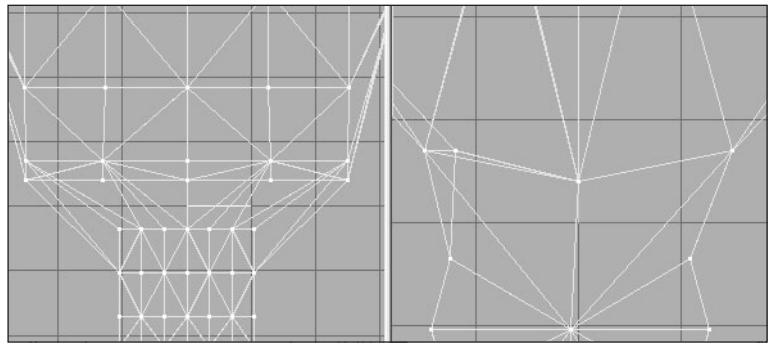


Figure 14.37 The well-aligned vertices.

You can now consider the torso to be finished. However, you can probably see areas where you can make obvious tweaks and adjustments. I did a few, just to make the integration of the back and the behind as well as the chest and the front abdomen a bit more natural looking. I also added a wee bit of "anatomical correctness," so to speak. Figure 14.38 shows the results of my tweaks. It should be fairly painless for you to duplicate these adjustments. The only operations I performed were Select (Vertex) and Move.

43. Save your mytorso.ms3d file so you don't lose all your work.

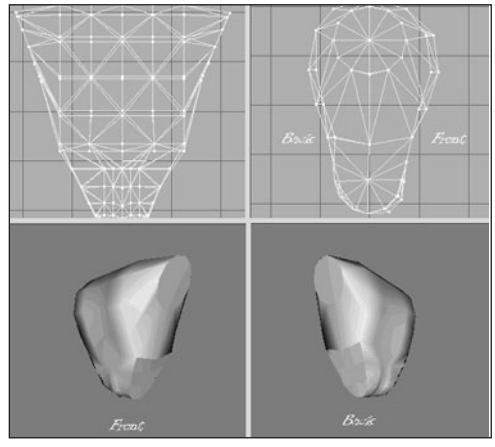


Figure 14.38 The final torso.

Matching the Head to the Torso

Now we should make sure that the torso and the head match correctly.

1. Choose File, Merge and browse until you find your head file, which should be at C:\3DGPAi1\resources\ch14\myhead.ms3d. Your head mesh will be loaded in, centered around the origin.
2. On the Groups tab in the toolbox, choose the new object (your torso mesh will be called "torso", so the merged head mesh will be the other one). Rename it as "head".
3. Click the Select button so that the head mesh is highlighted (and the torso mesh isn't), drag the head up in either the Front or the Side view, and position it as shown in Figure 14.39.

   I see two things I don't like right away—the head is bigger than it should be, and it also seems that its shape is a little too…ummm…*blah*. This isn't hard to fix, however.
4. Scale the head to 75 percent in the Y-axis only.

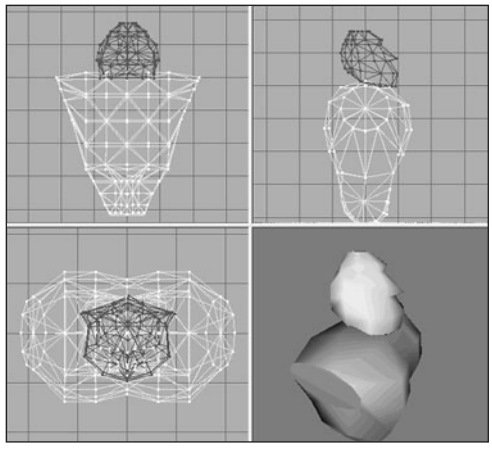


Figure 14.39 Matching head to torso.

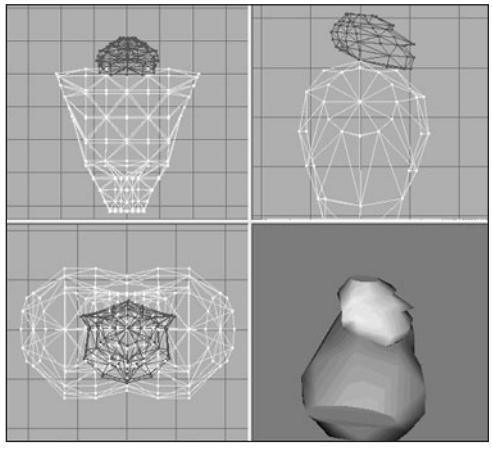


Figure 14.40 The reshaped head.

5. Move the head down until it just touches the top of the torso.
6. Rotate the head by 5 degrees around the X-axis so that the face is pointing a little bit down, as shown in Figure 14.40.

   There, that's more like it! Now to fiddle with the torso some more.
7. Select the vertices that form the shoulder sockets on both sides of the torso, using Figure 14.41 as a guide.
8. Scale the vertices to 60 percent in the Y- and Z-axes.
9. Save your work as C:\3DGPAi1\resources\ch14\mytorso.ms3d.

As we create the remaining parts of the model, we'll add them to this model by merging as we go along.

The Legs

When we start the legs, we'll want to keep the torso mesh around to use as a sizing reference, at least for the first little while. However, we won't need to have the head mesh in there, cluttering things up, so we'll get rid of that.

1. If you haven't already, save your torso as C:\3DGPAi1\resources\ch14\mytorso.ms3d.
2. Now save your file as C:\3DGPAi1\resources\ch14\mylegs.ms3d.
3. Delete the head mesh using the Groups tab in the toolbox.

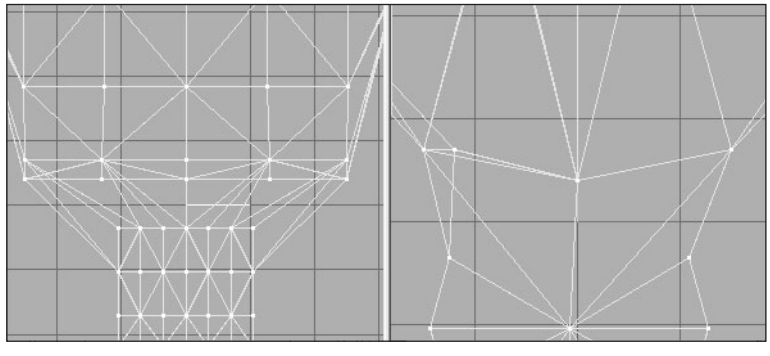


Figure 14.41 The shoulder socket vertices.

4. Select the torso mesh and drag it up about one torso's length above the origin.
5. Create a cylinder with 3 segments (stacks) and 12 slices (faces), and position and shape it as shown in Figure 14.42. This is the foot.
6. Create another cylinder and rotate it 90 degrees in the Z-axis, making sure that it is oriented so it runs left to right where the knee would be.
7. Using Figure 14.43 as a guide, move the vertices of the top of the foot up to meet the knee cylinder.

   By now you've probably realized that almost everything from here on is more a matter of style and taste, and less of technique. So you should feel free to go ahead and deviate from the specific construction details if you think of something you might like better.

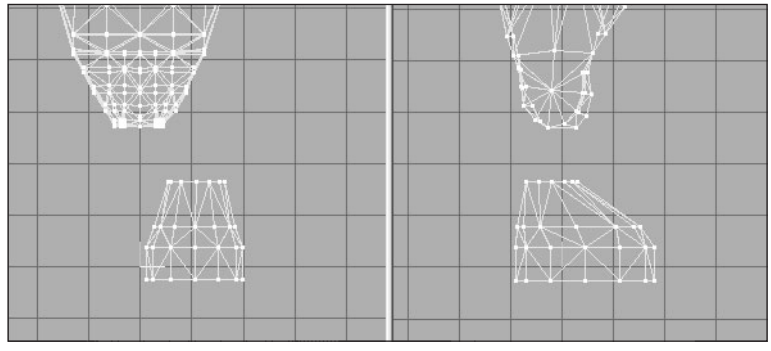


Figure 14.42 Shape and placement of the foot.

8. Reshape the knee cylinder as shown in Figure 14.44.

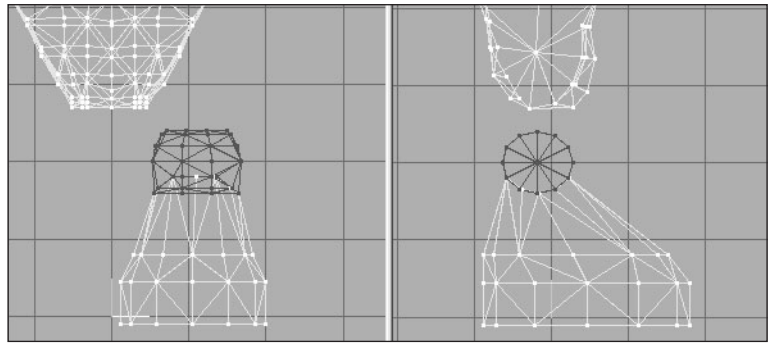


Figure 14.43 The knee.

9. Select the foot cylinder and rename it as "LeftFoot".
10. Create two more cylinders, and orient them as shown in Figure 14.44 to make the upper leg and hip.

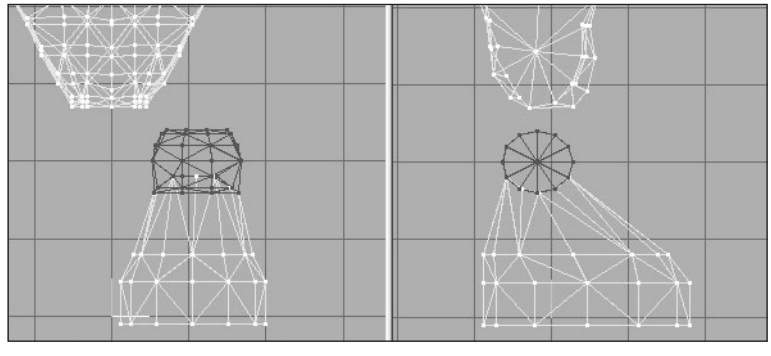


Figure 14.44 The left thigh.

11. Select the two new cylinders, plus the knee cylinder, and use the Regroup tool in the Groups tab of the toolbox. Name the resulting mesh "LeftThigh".
12. Shape the left thigh to match that shown in Figure 14.45—or to suit your own evil purposes.
13. With the left foot mesh selected, choose Edit, Duplicate. A duplicate of the leg is created in exactly the same location of the original, so you can't see it yet.
14. Choose Vertex, Mirror Left<--> Right. The duplicate leg mesh now appears on the other side, and in the right place, or pretty darn close.
15. Rename the new leg mesh as "RightFoot".

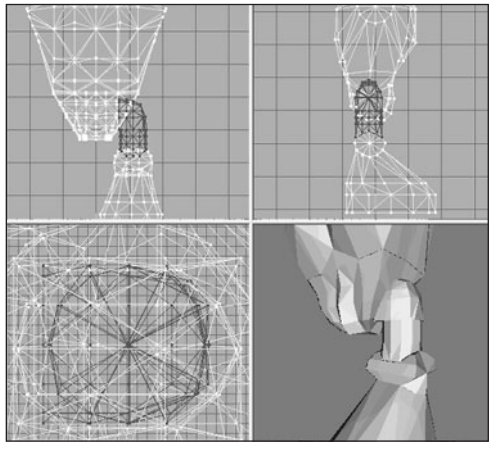


Figure 14.45 The finished left leg.

16. Now duplicate and mirror the left thigh in the same way, renaming the new thigh mesh as "RightThigh". You should now have two legs, each made up of a thigh mesh and a foot mesh and named appropriately.
17. Next, delete the torso mesh from the model. (You *did* save the torso in its own file earlier when I suggested it, right?)
18. Save your work! You should be saving this as C:\3DGPAi1\resources\ch14\mylegs.ms3d.

Integrating the Legs to the Torso

Just as we did with the head, we now have to integrate the legs with the rest of our model.

1. Open the file C:\3DGPAi1\resources\ch14\myhero.ms3d.
2. Select File, Merge, and choose the legs file you just created, which should be called C:\3DGPAi1\resources\ch14\mylegs.ms3d.
3. Choose the right leg, right thigh, left foot, and left thigh meshes and move them into position. You should now have a model pretty close to the one shown in Figure 14.46.

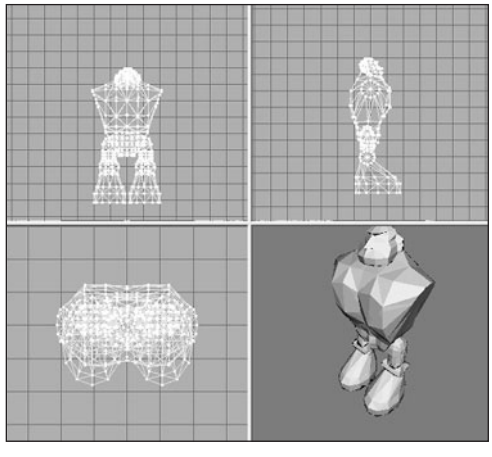


Figure 14.46 The Hero model with head, torso, and legs.

The Arms

Finally, the last set of meshes in the model. We can create the arms in exactly the same way we created the legs—building up from shape primitives, splicing them together until we have the desired mesh topology.

With arms comes the perennial question—what to do about the fingers? In some models we can make detailed meshes for each finger, with cylinders segmented at the knuckles, and so on. However, we must keep in mind that our goal here is to create a low-poly model, and that typically means fewer than about 1,500 polygons in the model. If we go over that count by a small amount, no big whoop, but we must remain mindful of it.

So, let's get to work!

1. Open your saved mytorso.ms3d file, and resave it as myarms.ms3d in the same location as your other work files.
2. Create a box offset to the left side (on the right in the Front view) of the torso, situated low near the bottom of the torso.
3. Duplicate this box and move the copy to abut the bottom of the original box.
4. Scale the second box to 80 percent.
5. Duplicate the second box and move the new box below the second.
6. Scale the third box to 80 percent of the second box.
7. Align the boxes as shown in Figure 14.47.
8. Hide the torso mesh to keep it out of the way for the moment.
9. Using Vertex Selection, the Move tool, and the Snap To Grid and Weld tools as you did with earlier parts of the Hero model, align the vertices of the three boxes as shown in Figure 14.48, and weld the vertices together.
10. Rotate the two bottom rows of vertices by −30 degrees in the Z-axis, as shown in Figure 14.49.
11. Move the two bottom rows in the Front view to align them as shown in Figure 14.50.
12. Rotate and move the bottom row of vertices to match what is shown in Figure 14.51.
13. Now, setting the Select mode to Groups, select all the groups of

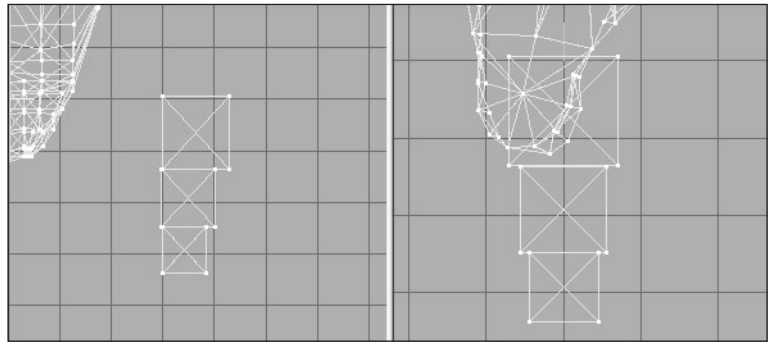


Figure 14.47 Alignment of the three boxes.

Team LRN

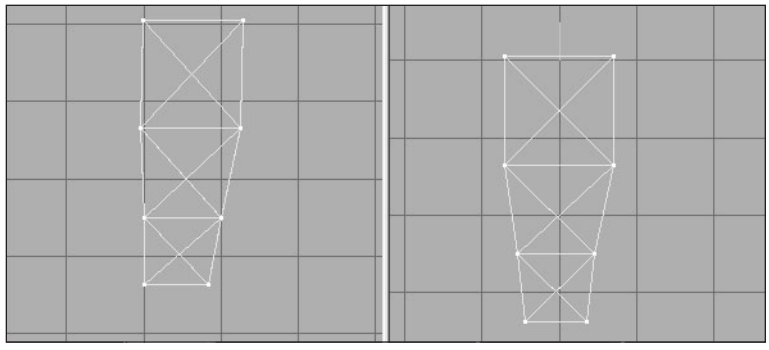


Figure 14.48 Welding the hand vertices.

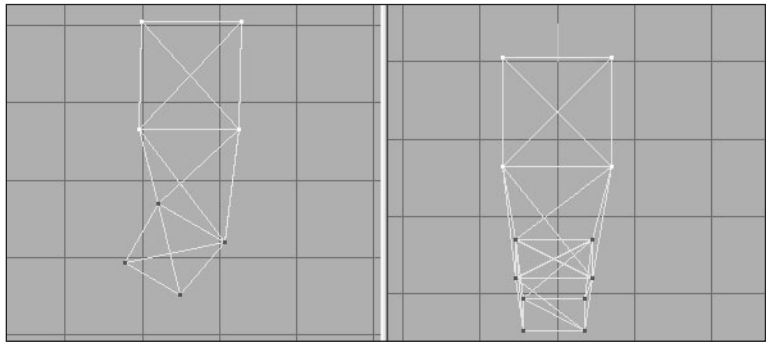


Figure 14.49 Rotating the two bottom rows.

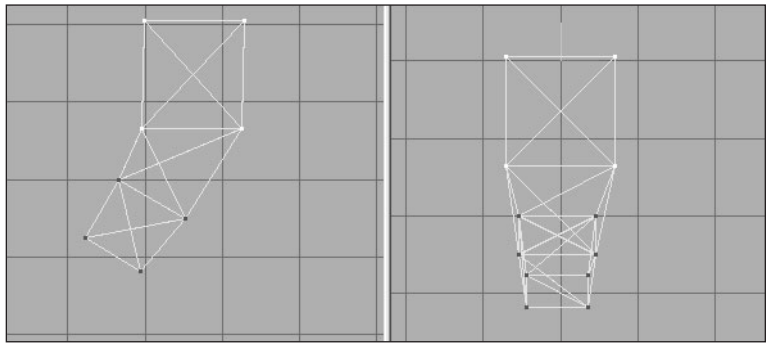


Figure 14.50 Moving the two bottom rows.

boxes, duplicate them, and move them forward to abut the front of the original boxes, using Figure 14.52 as a guide.

14. Now repeat the process in step 13, putting the new copies at the rear of the originals, using Figure 14.53 as a guide.
15. Now repeat the duplicating process *one more time*, but this time move the new boxes to the left side of the Front view. This is the thumb.
16. Choose Vertex, Mirror Left <--> Right to reverse the orientation of the thumb boxes (see Figure 14.54), and then scale the thumb to 50 percent.

17. Now we'll switch back to the hand part. In the Top view, select the vertices that are adjacent in the two forward parts of the hand boxes, as shown in Figure 14.55, panel A. Then choose Vertex, Flatten, Z and the vertices will be brought together onto a common plane, as shown in Panel B of Figure 14.55.

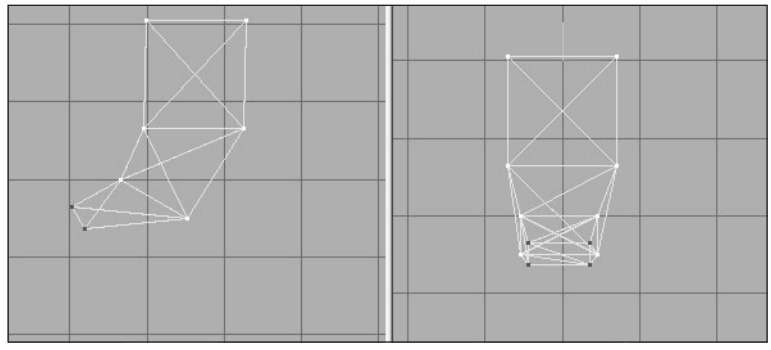


Figure 14.51 The bottom row of vertices.

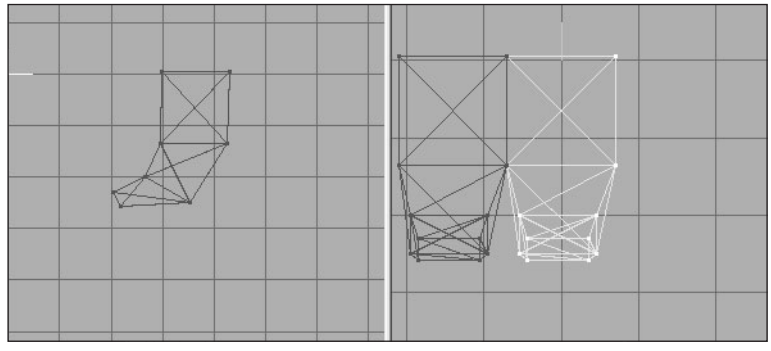


Figure 14.52 Two sets of shaped boxes abutting each other.

18. Double-check the other views, and if all vertices look to be coincident, then choose Vertex, Weld Together to weld them.

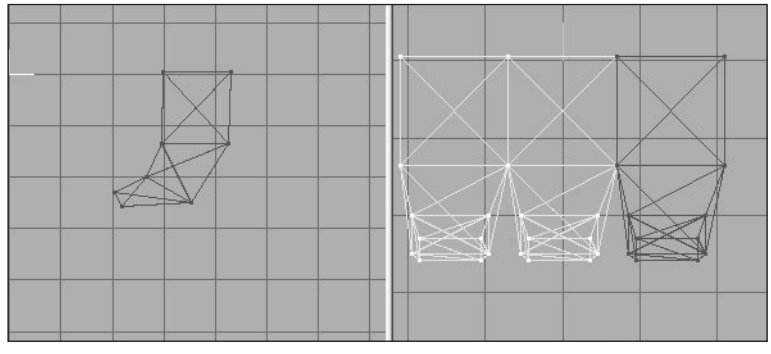


Figure 14.53 Three sets of shaped boxes.

19. Repeat step 18 for the rear part of the hand, welding the result. Compare with Figure 14.56 to make sure it is correct.
20. Select the rearmost vertices, in the baby finger area, and scale them to 50 percent in the X- and Y-axes, as shown in Figure 14.57.
21. Move the scaled vertices forward until they are close to the middle hand boxes, using the Top view in Figure 14.58 as a guide.
22. Repeat steps 20 and 21 for the front index finger area.

23. In the Front view, rotate and shape the thumb to approximate what's shown in Figure 14.59.
24. Unhide the torso and compare the size and positioning of your hand with the views shown in Figure 14.60. Rotate the hand to match, if required.

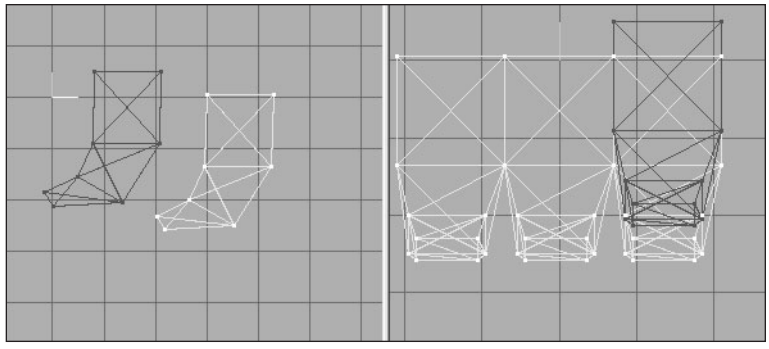


Figure 14.54 The start of the thumb.

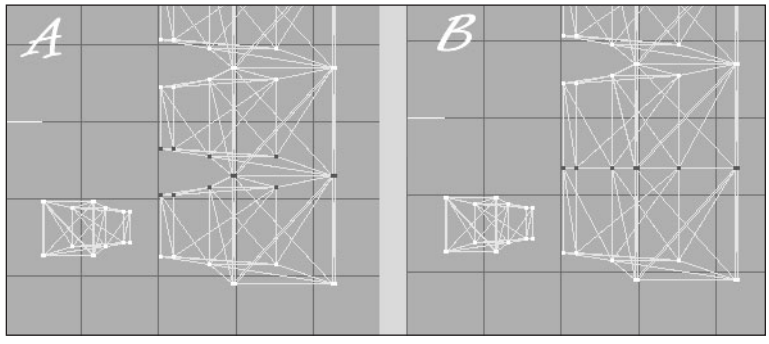


Figure 14.55 Welding the hand vertices.

Now you might be thinking that the hand looks awfully blocky compared to other parts of the model. You are right, but take heart. We can compensate for this with our skins. Remember, we want to keep our polygon count as low as we can.

25. Using the Groups tab in the toolbox, select all the hand groups, regroup them to form a new mesh, and rename it as "LeftHand".

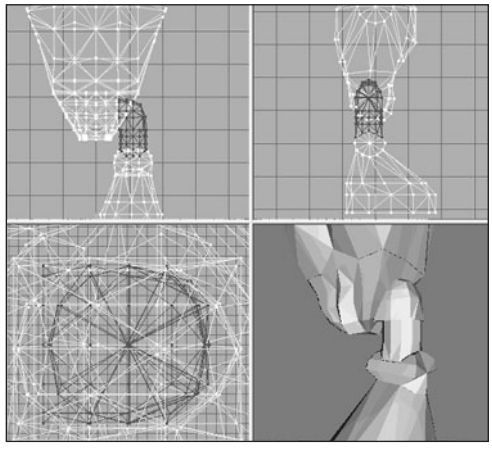


Figure 14.56 After the hand welding.

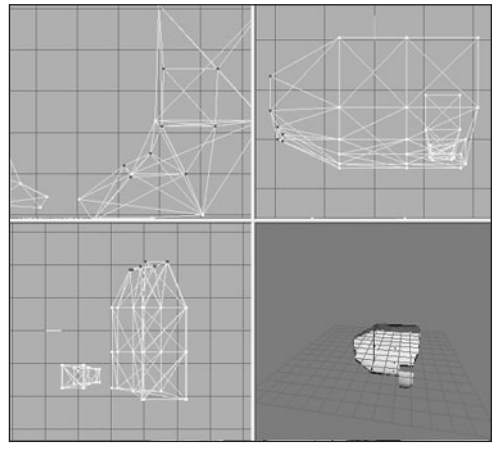


Figure 14.57 The scaled baby finger area.

26. In the Front view, use the Sphere tool set to four stacks and eight slices, and create a sphere that completely fills the left shoulder socket of the torso. Check all your views to make sure that you have it pretty close.

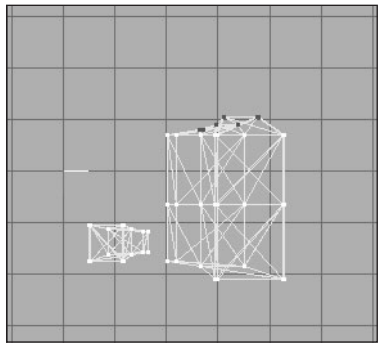


Figure 14.58 Placing the scaled baby finger vertices.

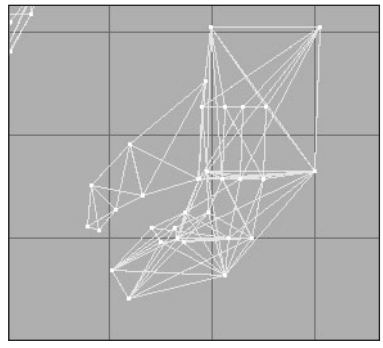


Figure 14.59 The thumb positioning.

27. Make another sphere with the same settings that fills the top of the hand, and place it there.
28. Make a one-stack, eight-slice cylinder that you can rotate and move into a position that connects the two spheres. Use Figure 14.61 as a guide for sphere and cylinder sizing and placement.
29. Select all the upper arm components, regroup them, and name the new mesh "LeftArm".
30. Select the new left arm mesh, duplicate it, and then choose Vertex, Mirror Right <--> Left.
31. Adjust the new mesh if necessary, and rename it as "RightArm".
32. Repeat the duplicating and renaming operations for the left hand mesh.
33. Delete the torso mesh.
34. Save your work! You should now have a pair of hefty arms that closely resemble those shown in Figure 14.62.

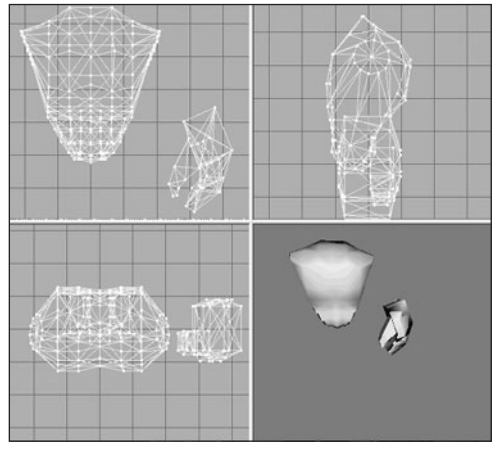


Figure 14.60 Comparison of torso with hand.

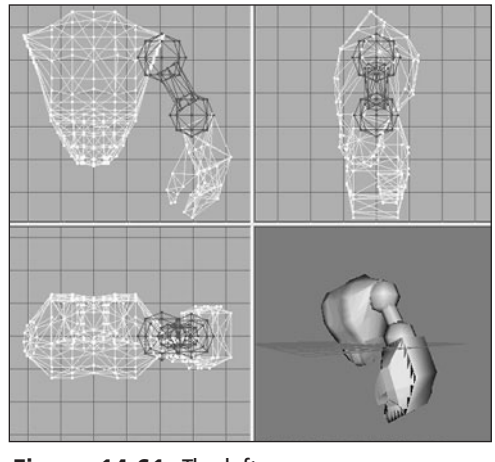


Figure 14.61 The left arm.

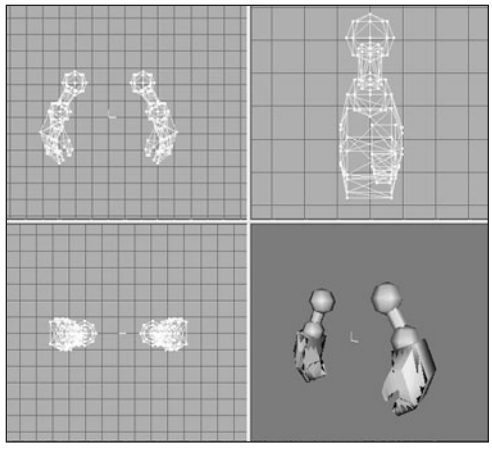


Figure 14.62 The completed arms.

Integrating the Arms to the Torso

Once again, it's integration time. If you are inclined to wonder about doing the model this way rather than all at once in one file, I want to point out that now you have a different source model file for each major component of your model. This allows you to make different mix-and-match models using the same components over and over. Just make three sets of arms, four sets of legs, five heads, two torsos, or something like that. Mix 'n' match 'em and you'll have all sorts of different model configurations!

1. Open the file C:\3DGPAi1\resources\ch14\myhero.ms3d.
2. Select File, Merge, and choose the arms file you just created, which should be called C:\3DGPAi1\resources\ch14\myarms.ms3d.
3. Choose both the right arm and the left arm meshes and move them into position. You should now have a model pretty close to the one shown in Figure 14.63.

The Hero Skin

Now it's time to skin the model. In Chapter 9 you learned how to create the textures for skins, and in Chapter 13 you learned how to do simple UV mapping for skins. Next, we have to do the UV mapping for the player-character, which is somewhat more complex. We are not going to go over the creation of the texture for Hero character skin. The Resources folder includes a mapped Hero skin texture for you to use, but I encourage you to make your own in the same fashion as the one for the Standard Male.

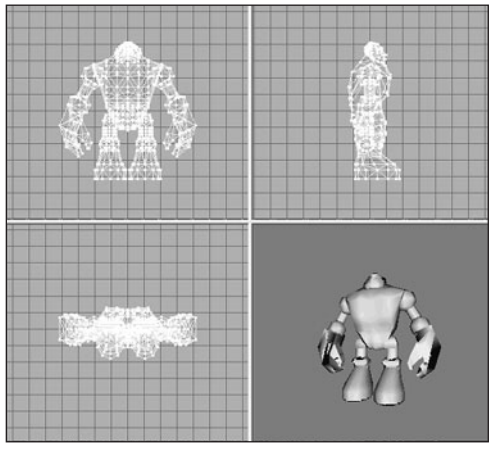


Figure 14.63 The completed Hero model.

1. If MilkShape is not already running, launch it and open your Hero model, located at C:\3DGPAi1\resources\ch14\myhero.ms3d.
2. Choose File, Export, Wavefront Obj and export your Hero model as C:\3DGPAi1\resources\ch14\myhero.obj.
3. Launch UVMapper and maximize the window.

4. Load the C:\3DGPAi1\resources\ch14\myhero.obj. You will see a crazy quilt of lines. This is the "default" mapping created by MilkShape. Let's forget about that, because we are going to create our own mapping.
5. Choose Edit, New UV Map, Planar and then use the settings shown in Figure 14.64.
6. Choose Edit, Settings, click Color by Group and then click OK. Your screen should now look like Figure 14.65.
7. First, choose Edit, Select All and press the forward slash ("/") key on the numeric keypad to shrink the selection to 25 percent, half-sized in the x dimension and half-sized in the y dimension (the asterisk ["*"] on the numeric keypad will do the opposite). Press Enter to save your adjustment. If you didn't like the adjustments you just made, press Esc to undo your changes since making your last selection or remapping.
8. Choose Edit, Select by Group, choose the group "head", and then click OK.
9. Choose Edit, New UV Map, Spherical. Use the Spherical Settings shown in Figure 14.66.
10. Press the equal sign ("=") on the numeric keypad to expand the head selection to fill the window, and then press the numeric keypad "/" a few times to shrink the selection. Use your mouse to drag the head to the upper center of the window, as shown in Figure 14.67.

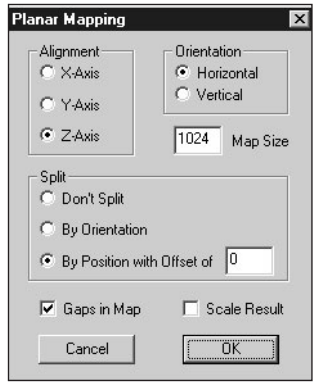


Figure 14.64 Planar mapping settings for Hero model.

Now you might notice that there appear to be two triangles out of place in the head unwrapping. Look inside circle A in Figure 14.67 and see if you can spot them. In your model this may not be the case, but the more closely your model matches the one I've done here, the more likely this is to happen. This little oddity is easily fixed. You should be able to do the mapping without that happening—it's all a matter of which settings you use. You can

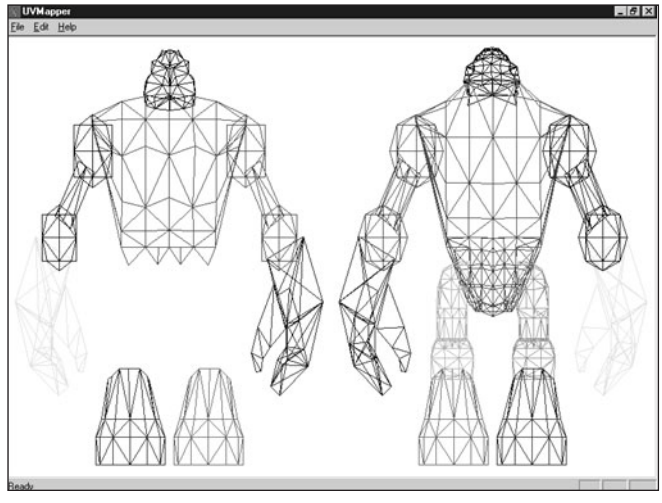


Figure 14.65 The unwrapped Hero model.

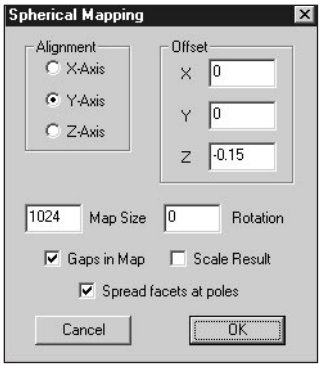


Figure 14.66 Settings for the spherical mapping of the head.

try to get the right settings by trial and error. However, the simplest fix is to just move the miscreants to their lawful location. So that's what we'll do.

11. Drag your cursor over the middle of the two wayward triangles. Don't touch any parts of any other triangles in any other part of the model. The triangles will now appear surrounded by a selection box.
12. Click and drag them over to the right-hand side where there is that suspicious-looking gap, and place them as well as you can. Location B in Figure 14.68 shows where the triangles end up.
13. Use the arrow keys to adjust the position of the triangles. There you go!

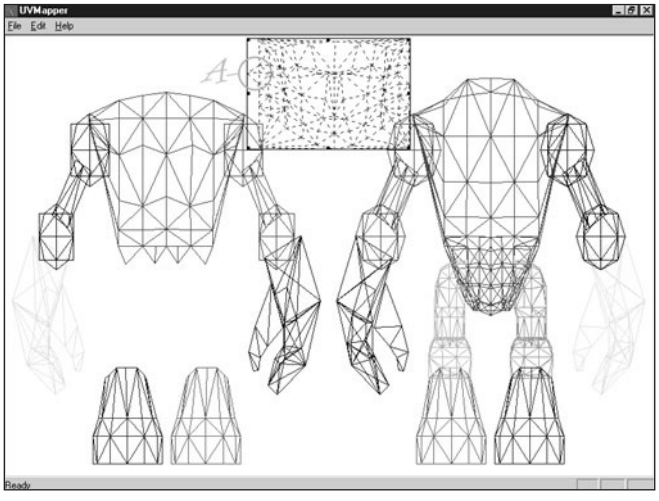


Figure 14.67 The unwrapped head.

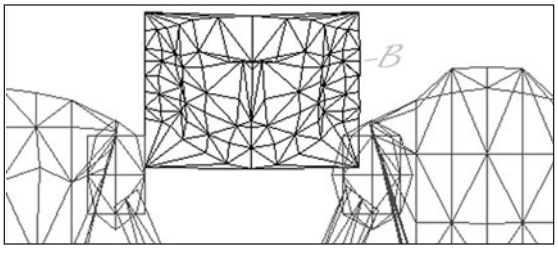


Figure 14.68 The adjusted triangles.

Now you need to do a bit of housekeeping-like fiddling.

14. Choose Edit, Select, All. You will get everything on the screen selected in a selection box with the little black sizing handles at the corners and midway along each side.
15. Grab the sizing handle on the right side. Your cursor should change to the left-right sizing cursor (this is an arrow pointing left and right).
16. Drag the sizing handle toward the left until you get a blank space on the right a little wider than the width of the head.
17. Choose Edit, Select, Group and choose the head. Drag it over to the upper right, in the blank space you just created. You should now have a layout like Figure 14.69.

Team LRN

As you work you will probably reorganize your layout a few times—that's perfectly normal. You want to keep it clean and make sure your items are easily selectable.

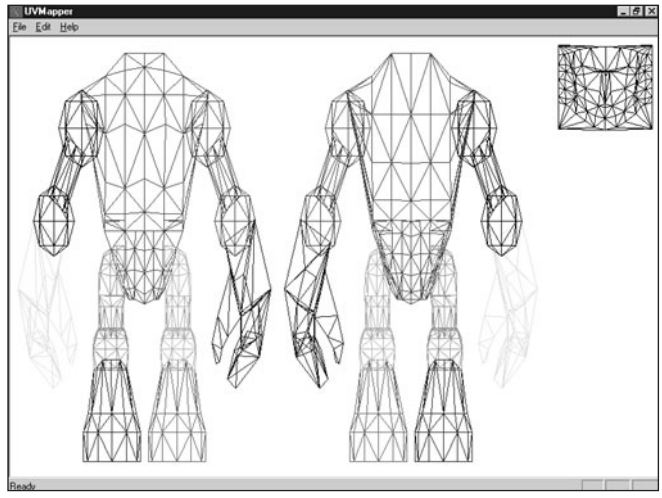


Figure 14.69 The reorganized map.

18. Now choose Edit, Select, By Group, and choose the LeftHand group.
19. Choose Edit, New UV Map, Box. You will get the Box Mapping dialog box, as shown in Figure 14.70. Make sure you have Split front/back turned off, and Gaps in Map turned on. Click OK.

    The unwrapped left hand will appear in the window, surrounded by the selection box.

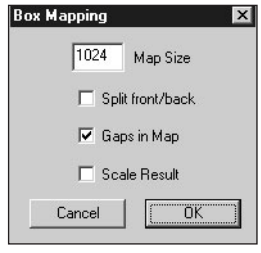


Figure 14.70 The Box Mapping settings.

20. Move and size the hand mapping, placing it in the center of the window in the blank area. Make sure it is small enough to allow the mapped right hand in here as well (see Figure 14.71).
21. Perform the same UV mapping operation and placement operation on the RightHand group, putting them in the center space.
22. Next, map the left and right feet the same way. For each group as you unwrap it, size the sole (oval shape) so that it is longer than it is wide. Place the feet underneath the main model, as shown in Figure 14.72.
23. Next, map all the arms and legs. Use Planar mapping for these.

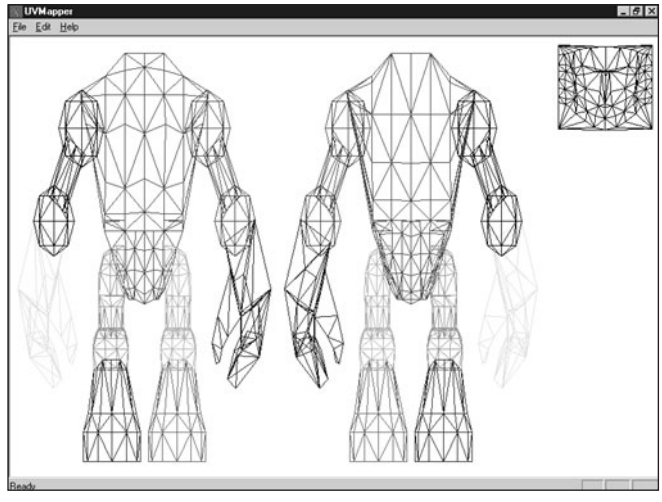


Figure 14.71 The UV mapped hands.

24. Shrink and move the mapped objects, ensuring that no mapped objects overlap any others.
25. Once all the objects are mapped, overlap similar items (with the exception of the torso front and back), and enlarge the torso, hand, and head objects as much as possible. The larger the mapping, the more detail can be applied in the texture.
26. Finally, move and arrange the objects to match the layout in Figure 14.73. This is the final texture-mapping layout that we'll use for the template.
27. Choose File, Save Model and save it as C:\3DGPAi1\resources\ch14\myhero.obj (thereby obliterating the one you created from MilkShape; don't worry though—you can always export another from MilkShape if needed).

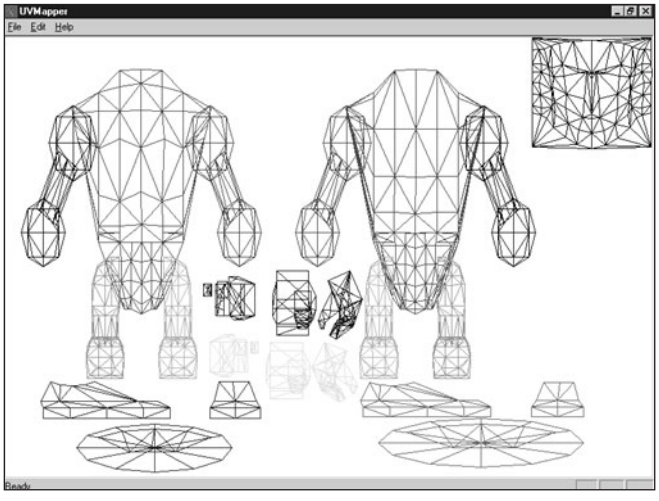


Figure 14.72 The UV mapped feet.

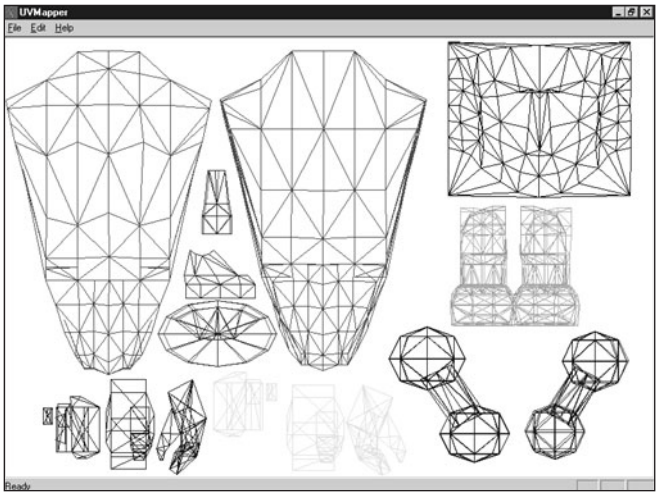


Figure 14.73 The final UV mapping layout.

28. Choose File, Save Texture Map and save the map as C:\3DGPAi1\resources\ch14\myhero.bmp. Make sure the texture size is set to 512 by 512.
29. Launch MilkShape and create a new file.
30. Choose File, Import, Wavefront Obj and import the C:\3DGPAi1\resources\ch14\ myhero.obj file.
31. In the Materials Tab of the toolbox, click New.
32. Click the top Texture button and locate the C:\3DGPAi1\resources\ch14\myhero.bmp texture map template file you created in UVMapper.
33. Rename the material as "heroskin".
34. Using the Groups tab, select all the meshes and then switch back to the Materials tab and click Assign. You should now have a 3D view that resembles Figure 14.74.

Of course, your version is in color. The lines of the triangles in the color groupings assigned in UVMapper are clearly visible. You are now in a position to go ahead and use Paint Shop Pro to create your skin for the Hero model. Refer back to Chapter 9 if you need a refresher. Make sure to save your skin as a JPG file type if you want to minimize file size. This means that you will also have to go back into MilkShape and redefine your material to point at the JPG version and not the BMP version.

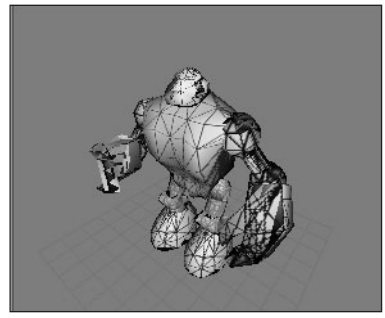


Figure 14.74 The 3D view showing the UV template texture.

We'll continue from here into the animation section using the UV template hero skin that I've included in the C:\3DGPAi1\resources\ch14\hero.jpg file.

Character Animation

Well, a static model, no matter how cool looking when it's standing there, is not terribly interesting in a first-person shooter. We're going to have to animate that sucker!

If we were a big name, big money shop, we might go out and hire a motion capture studio to make our animations. But we're not—we're indie developers! So we will have to explore other options, and there are some.

On the Internet you can find some stop-motion photography sequences that might help you develop your character animations. There are also freely downloadable files with character animations in them—they will probably have a different skeleton structure than the one we use here, but a certain amount of tweaking can go a long way.

I know someone who manually creates animations using action figures that he poses, changing the poses step by step as he works through the animation, converting what his eye sees into the appropriate frame in his animation program. This is certainly a good low-budget option.

In this book we are going to hand-build our animations, because the point is to learn how to do it. They may not be the best animations in the world (or maybe they will be!), but they will be your very own if you make them yourself. If you need a model, ask a friend or family member to step slowly through whatever it is you are trying to animate, if it's humanly possible. You'd be surprised how helpful that can be. Bribe them with their favorite dessert or something.

Animating Characters in Torque

The general method for making animated characters for use in Torque is to construct a skeleton that corresponds to components of the model and then attach that skeleton to

the corresponding components in a process called *rigging*. We then create a sequence of *keyframes*—essentially a series of skeleton poses. When the Torque Engine wants to animate the model, it calculates the positions of the meshes in the model by the position and rotation of the nodes (the joints where the "bones" of the skeleton connect to each other) based on where the keyframes appear in the animation timeline.

We are going to create six different basic animations:

- root (same as idle in some systems)
- run
- look
- head
- headside
- death

Table 14.1 Torque-Supported Animation Sequences

Sequence Name	Description
root	This is the basic "not doing anything" animation—usually the character is standing and fidgeting.
run	This is the animation used when the character is running forward.
walk	When the character's speed is less than running speed, he walks, using this animation.
back	This is the animation used when the character is running backward.
side	This is the animation used when the character is running sideways, sometimes called *strafing*.
look	A simple animation where the character's right arm points where the character is looking, such as when holding a weapon.
head	The head looks up or down depending on where the character is looking.
fall	This is the pose of a character, or its animations, when the character is falling off a cliff or building.
land	It's the sudden stop at the end of a fall!
jump	This is a jump made while running.
death1	Like it says. One of 11 possibilities. You don't need them all, but like they say, "Variety is the spice of life, er, death, or something."
death2-death11	Ditto.
looksn	This is the same as "look", but with weapon held close.
lookms	This is the same as "look", but with arms loose.
scoutroot	This is the animation used when the character is astride something like a motorbike.
headside	This is the animation used when the character is turning its head from side to side.

Team LRN

light_recoil	This is the weapon recoil, used to show the character's reaction to firing a weapon.
sitting	This is the animation used when the character is seated, like sitting in a car.
celsalute	This is an animation, activated by Ctrl+S as default, an in-game salute or taunt animation. You can use it for whatever you want.
celwave	Activated by Ctrl+W as default, this is another in-game salute or taunt animation.
standjump	This is another jump animation, but this time from a standing pose, like "root".
looknw	This is another weapon "look" with a variation of the arms loose pose.

These animations correspond to character animation support built into Torque. The names must match the names used by Torque; however, we can add our own arbitrary animations and activate them from within the script programs if we want. There are also other animations that Torque supports that we won't cover here.

note

Torque supports animating in several ways, one of which we are covering in this book, where the animation is embedded with the model in the DTS file. Another system Torque can use is the DSQ or Torque Blended Animation Sequence system, which has two important features: It supports *blended* animation, where two different animations are played for the same model at the same time, and it supports the separation of animation sequences from the model (DTS) using the sequence files (DSQ) format. Unfortunately, the MilkShape exporter for Torque does not support exporting sequence files. This might change in the future.

In Figure 14.75 the highlighted (black lines) sphere is a joint, or node. The spike between the two spheres is a "bone." The direction the spike is pointing indicates the relationship between the nodes. The node at the big end is the parent, while the other end is the child node. Notice that in Figure 14.75 the parent node is rotated 60 degrees between frame A and frame B, and the child node follows the rotation. The unattached node doesn't move. Note that the horizontal and vertical lines inside the nodes are angled in the rotated nodes, but not in the unattached node.

Open your myhero.ms3d file, if it isn't already open. Set Joint Size to 3.0 or 4.0 in the Preferences dialog box on the Misc tab. We need to use such a large joint size because the scale of the Hero model is quite a bit larger than the Standard Male from Chapter 9.

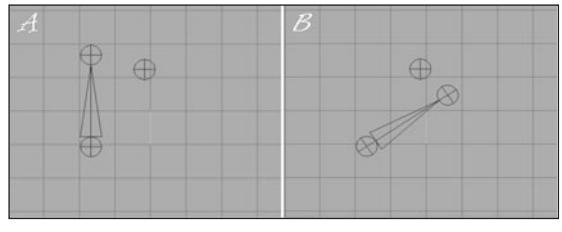


Figure 14.75 Bone movement during joint rotation.

Building the Skeleton

Before we can create the animations, we need to construct the character's skeleton. We build the skeleton from the "bottom up" so to speak, beginning with the base node, and working toward the outer extremities.

1. We will start with the base node (or joint), which we place at the origin: (0,0,0).

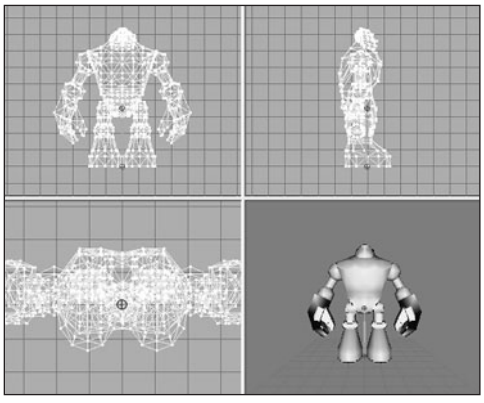


Figure 14.76 Placing the base and pelvis nodes.

2. Now make sure that the base node is selected, and then place another node roughly in the groin area. Figure 14.76 shows the relative appearance of these nodes. Make sure that the big end of the bone that joins these two joints is at the end where the base node is.
3. Rename the new node as "pelvis".
4. Add all the new nodes and label them appropriately. Figure 14.77 provides a guide to the node placement and their names.

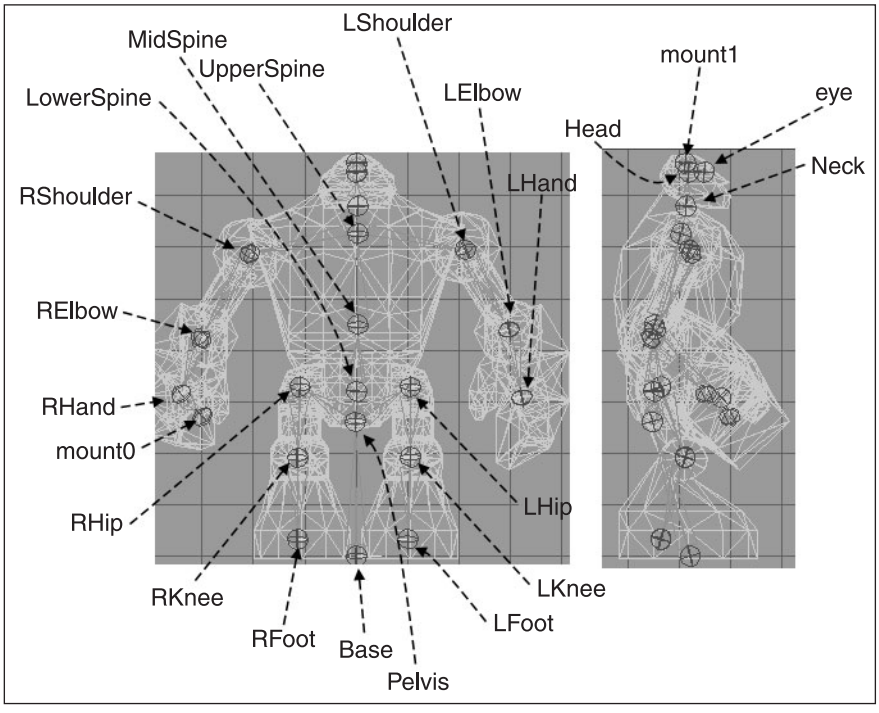


Figure 14.77 The Hero skeleton with labeled nodes.

5. Adjust the mesh rotations to match the pose in Figure 14.78. You want the character slightly bent at the knees, with his left arm slightly bent beside him and the right arm bent up at 90 degrees. Starting with the base node and moving on to the pelvis, hips, and shoulders (in that order), adjust the joints of the skeleton. Remember to rotate hip, knee, elbow, and shoulder joints to move the joints at the extremities. Pay particular attention to the placement of the pelvis, lower spine, and hip nodes. The pelvis should be well below the level of the hip nodes, and they should be just a tad higher than the lower spine.

note

The node names and bones you see in Figure 14.77 are not in any way standard. They don't have to be. However, there are standard skeletons for other games that you can use. Torque also has a standard biped skeleton that is the same as the one used for CounterStrike. But we can't really take advantage of it, because it is only useful with sequence files, which we do not yet have the ability to use given the current DTS Exporter that is available. So we may as well use our own skeleton.

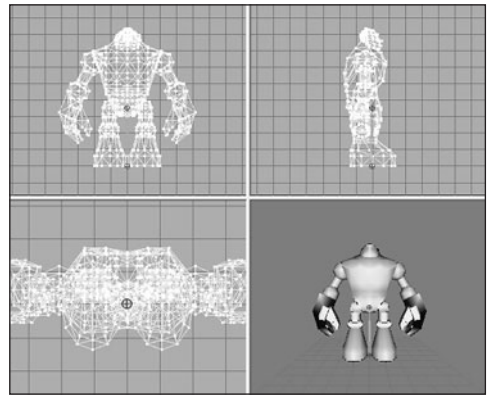


Figure 14.78 The pose-adjusted skeleton.

Rigging: Attaching the Skeleton

Now we have the skeleton built, the nodes have been named, and the bones are aligned into a pose we like. Next we are going to attach our model to the skeleton. That way, when the skeleton is manipulated, the mesh of the model will follow suit. It is during this step that you might be inclined to thank me for insisting that you retain mesh groups for the different model components like arms and feet and so on!

Rigging the Head

We'll begin with the head, just to get a feel for the rigging operation.

1. In the Joints tab in the toolbox, choose the joint (or node) named "head". Make sure it appears highlighted in red in the wire-frame views.
2. Switch to the Groups tab and choose the head mesh. It should appear highlighted in red, as you already know.
3. Switch *back* to the Joints tab and click Assign. Now the head mesh is assigned to the head node. To double-check, just click anywhere in a blank space in a wire-frame view to make sure that no objects are selected, choose the head joint to ensure it is selected, and then click the SelAssigned (Select Assigned) button. The head mesh should appear highlighted. If not, go back and repeat these three steps.

Aw shucks, there it is—the head is now rigged! Of course, that's not the end of the story. There is still the rest of the model.

There's also the issue of what to do when a bone is rigged wrong. Sometimes it's a trivial fix, and other times you might have to rerig the whole model. Or you might have to rig a model by attaching a node to just a few vertices rather than a whole mesh or submesh. That can get very, um, *fiddly*—I guess that would be a reasonable description.

Part of the simplicity in rigging this model comes from the technique we used; building from primitives allows us to easily define meshes and submeshes. We'll use a "one node per mesh" rule of thumb. It can get trickier using other techniques, but sometimes those other techniques might be more appropriate for the model you want to build. It's a judgment call, as everything important tends to be. There is one exception—as there always is—to the "one node per mesh" rule that we'll get out of the way next.

Rigging the Torso

Okay, so the head was a cakewalk. It wasn't even necessary to show any pictures for you to be able to follow along. How about the torso then—duck soup again, no?

Well, yes…I mean *no*, actually. No duck soup for this one!

The head mesh is attached to the head node, and that is fine. Tilting or rotating the head node will indeed move the head in the manner we want. There really isn't a whole lot to choose from. The neck is more a part of the spine than the head. The camera and mount1 nodes aren't even related to the skeleton. They are *special nodes* that will have a different role to play in Torque, which we'll cover later. So that leaves the head node to control the head mesh.

The torso, though, has at least five nodes that it might be attached to. But which should it be? Let's eliminate the neck node for now. That leaves the spine nodes or the pelvis node. Actually we *can* use more than one node for a mesh giving different parts of the mesh to different nodes. When we built the torso mesh, we actually combined two primitives together, remember that? One was the chest cylinder, and the other was the abdomen cylinder. We could have left them as two separate submeshes, but I wanted to show you how to join them together. We can still use them as if they were two separate meshes, by assigning their respective vertices to different nodes.

If you look at the nodes, you'll see that the pelvis node is pretty well the obvious candidate to control the abdomen part of the mesh. The upper spine node, although probably not as obvious, is likely the best candidate for the other node, because it exists in circumstances similar to the pelvis—there are limbs attached. So we'll go with these two and see how that works out.

Team LRN

What this will mean in terms of animation is that we can have the vertices that are attached to one node move in one way, while the vertices attached to the other node move differently. Or not. It all depends on how you rig it.

None of this is strictly necessary. The animations we are going to create don't actually *require* the torso mesh be given more than one node, but it's a good thing to learn, so we'll do it.

1. To get started with rigging the torso, let's tidy up the ol' drawing board a bit by hiding all the meshes except the torso mesh. If you've forgotten how, just go to the Groups tab of the toolbox, choose each mesh, and click Hide. Unfortunately, we can't selectively hide parts of the skeleton. It's either all or nothing when it comes to the bones.
2. Choose the pelvis node in the Joints tab.
3. Switch to the Model tab and set the Select tool to Vertex mode. Then select the vertices that are the abdomen. You can use either the Front view or the Side view. Figure 14.79 shows the vertices to select.
4. Back in the Joints tab, click Assign. Now the vertices are attached to the pelvis node.
5. Now choose the upper spine node, and then select the vertices for it, using Figure 14.80 as a guide.
6. Click Assign in the Joints tab, and that should do it.
7. Double-check to make sure you didn't overlook any of the vertices by choosing each node in turn, clicking the SelAssigned button, and looking to see which vertices for that node might have been missed. If you did miss any, you can simply select the node, select the vertices, and then click Assign to add them to the nodes list.

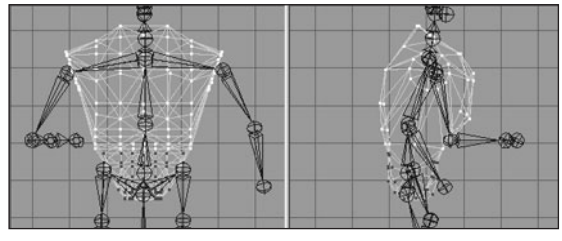


Figure 14.79 The abdomen vertices.

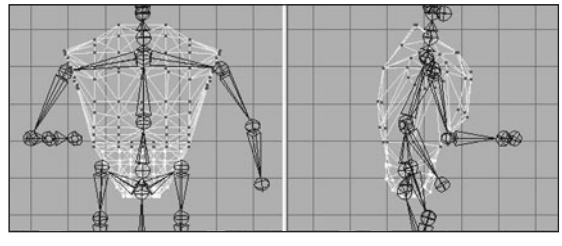


Figure 14.80 The chest vertices.

There, that's the torso. It might not have seemed so difficult a task to you, but to me it was a nightmare! Well, maybe not that bad, but it shows you the kinds of decisions you will have to make when rigging your models. What goes where and how best will it work?

Now that we have a few nodes rigged, let's take a look and see what they actually do.

Team LRN

1. If you don't have a button at the lower right called "Anim," then choose Window, Show Keyframer, and make sure there is a check mark there.
2. Click Anim to activate the Keyframer.
3. Using the Select tool in the Joint mode, select the pelvis joint (or you can use the Joints tab to do the selection).
4. Use the Rotate tool in freeform mode in the Right Side view. You will recall that freeform rotation is a simple matter of selecting the Rotate tool, then clicking in the wire-frame view, and dragging the cursor left and right.

Now what you should be seeing is the entire torso, plus the head, rotate around the pelvis. You should also see some strange things as well. The arm, leg, foot, and hand meshes don't move. That's because they aren't rigged yet.

But notice that the leg bones are rotating when you rotate the pelvis. Aha! I don't know about you, but when I bend over, my legs don't move back. Well, not unless I'm floating in water, of course. So the pelvis node, while it seems to be an obvious candidate for bending your character at the waist, looks to not be the right one.

So go back, right now, and change it. It's simply the same procedure I showed you for the pelvis, except you do it for the lower spine node. Make sure to click the Anim button to take it out of the Keyframer first, or you won't be able to make the changes. I'll wait.

Musical Interlude.

There you are. Now that that's done, go back into the Keyframer as I showed you before, and check the rotation of the lower spine node.

Another Musical Interlude.

Good! So everything should be working as expected now. The torso and the head meshes bend over in unison, and all the bones attached above the lower spine bend in unison, as shown in Figure 14.81. As you've probably deduced, it is now a reasonably minor matter to rig the rest of the nodes. Use Table 14.2 to guide you in your rigging.

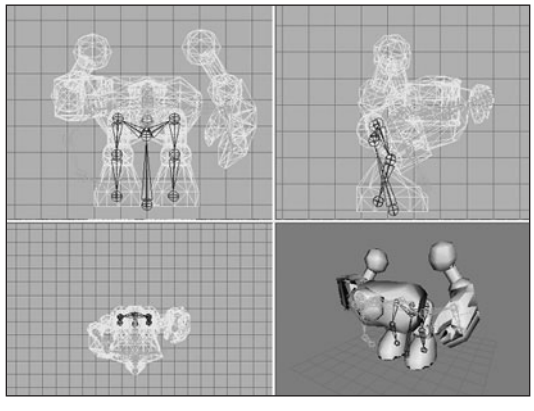


Figure 14.81 Bending at the lower spine.

You just need to match a mesh to a node, attach it, and move on. I'm enjoying the music here, so you go ahead and do the rest of the rigging, and I'll sit back and relax.…

Yet Another Musical Interlude.

Great! With that done, let's move on.

Table 14.2 Hero Rigging

Node	Mesh to Be Rigged
ZHead	Head
UpperSpine	Torso—chest-area vertices
LowerSpine	Torso—abdomen-area vertices
LShoulder	LArm
RShoulder	RArm
LElbow	LHand
RElbow	RHand
LHip	LThigh
RHip	RThigh
LKnee	LFoot
RKnee	RFoot

Idle Animation

The idle animation is the one used by Torque when the character is just standing there, doing nothing in particular. In some games you will see some pretty complex idle animation where the character scratches himself in rather inconvenient locations, looks around, scuffs his feet, and so on. We're just going to do a basic breathing sequence so that you'll know that the character is alive. The name for the idle animation in Torque is root.

Even with a basic animation, the watchword is *subtlety*. Don't overdo it.

1. Make sure the Keyframer is enabled by clicking the Anim button in the lower-right corner.
2. Set the number of frames in the Keyframer to 30. Do this in the right-hand edit box in the lower-right corner of the Keyframer (see Figure 14.82).
3. Move the slider to the 1st frame.

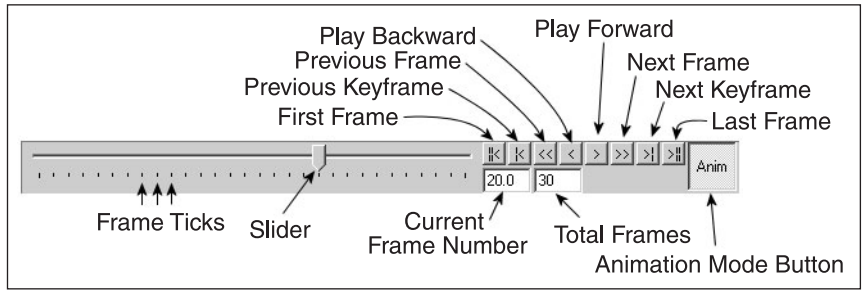


Figure 14.82 The Keyframer control panel.

4. Choose Animate, Set Keyframe. This indicates that this particular frame is a keyframe.
5. Move the slider to the 15th frame.
6. Take note of the angle of the elbows and hands.
7. Select the midspine node and rotate it 5 degrees around the X-axis.
8. Rotate each of the elbows in the opposite direction by about 5 degrees to place them back where they were before.
9. Choose Animate, Set Keyframe to set the keyframe attribute for this frame.
10. Move the slider back to the 2nd frame.
11. Choose Animate, Copy Keyframes.
12. Move the slider to the 30th frame.
13. Choose Animate, Paste Keyframes.
14. Save your work!

Figure 14.83 shows the subtle pose difference between the 1st and the 15th frames. Now you can test your animation by clicking the Play button on the Keyframer controls. The Play button is the one that looks like a single arrow pointing to the right.

As long as the Play button is down, the animation will loop. If you find it runs too fast, you can change the FPS number in the Preferences dialog box to a lower value to slow the animation.

Notice that when the animation is actually running, that subtle pose change becomes quite noticeable.

tip

An excellent tool called characterFX is useful for creating animations, and it works well with MilkShape. Unfortunately, for logistical reasons it could not be included with the tools on the companion CD. However, it does a great job of streamlining the process and is flexible, so a quick Google search for it on the Internet might be worth your while!

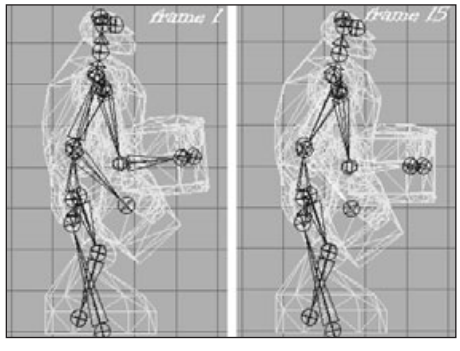


Figure 14.83 The difference in poses.

Run Animation

The run animation is the staple of first-person shooters. Run and shoot, run and shoot. Our Hero character has a somewhat awkward lower body, which will tend to make any animation of him running look a bit goofy. Well, we'll turn that into a feature and capitalize on that goofiness.

1. Set the Keyframer to 120 frames. Additional frames will be added after the 30 you started with for the idle animation.
2. Move the slider to frame 31.
3. Make sure that Operate on Selected Joints Only in the Animate menu is enabled.
4. Rotate the right hip joint slightly and then choose Animate, Set Keyframe. The pose should be similar to the resting pose. The reason why we touched that hip joint at all is to ensure that there is at least one joint in the frame that was affected so that a keyframe will be made. You should have a pose much like that shown in Figure 14.84.
5. Move the slider to frame 40.
6. In the Right Side view, select the Base joint, and move it forward by one grid square and up about three-quarters of a grid square, as shown in Figure 14.85. This movement of the base joint moves the entire model—it's a transformation across the ground. This transformation is necessary in order to notify the Torque Engine that the model is moving and how fast it is moving. It doesn't need to be precise, but it does need to be there.
7. Select the right hip, and then in the Side view rotate it so that the leg moves forward.
8. Rotate the right knee forward as well, until the leg matches the configuration in Figure 14.85.
9. Repeat the rotations for the left leg and move it backward. In order to get things looking right, you might have to adjust the joint positions slightly by moving them, but not by much.
10. Rotate the left arm using the left shoulder node and the left elbow node, swinging the hand forward until it is approximately opposite the right leg, as shown in Figure 14.85.
11. Set frame 40 to be a keyframe.
12. Move the slider to frame 50. Use Figure 14.86 as the guide for this frame.
13. Move the base node one more grid square to the right, but this time move it back down to the ground level.

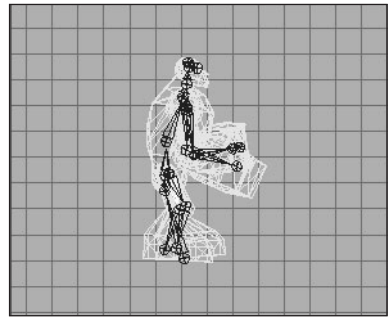


Figure 14.84 Frame 31.

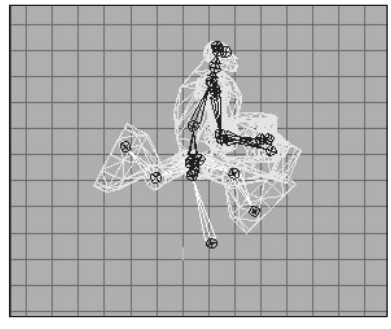


Figure 14.85 Frame 40.

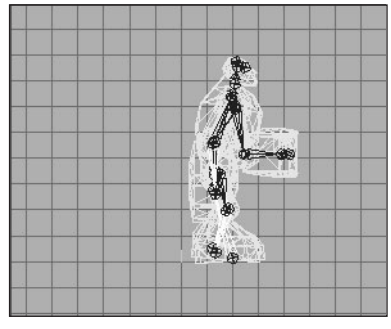


Figure 14.86 Frame 50.

14. Move all of your legs and joints back to approximately the same configuration as in frame 31.
15. Swing the left arm down to the side of the model.
16. Set this frame (50) to be a keyframe.
17. Move to frame 60. Use Figure 14.87 as the guide for this frame.
18. Pose frame 60 the same as frame 40, except swing the legs in the opposite directions.
19. Swing the left arm back and rotate the elbow so that the left hand comes up parallel to the ground.
20. Set this frame to be a keyframe.
21. Move to frame 70. Use Figure 14.88 as the guide for this frame.
22. Swing the arms and legs back to roughly the pose they had in frame 31.
23. Set this frame to be a keyframe. Use the Play Forward button to watch the animation. If the animation seems to be too fast or too slow, change the FPS setting in the Preferences dialog box until it seems right, and take note of the value you use.

Now you have probably noticed that although we set the pose in only five frames, the program automatically *interpolated*, or figured out, what the in-between frames should look like. Torque does the same thing for us when we use the model in-game. This is goodness. That's as much of the run animation as we're going to do here, but you should practice working with this for a while. The first place you should start is to set the keyframe exactly in the middle of the ones we've already set—at frames 35, 45, 55, and 65—and adjust the leg positions to get a better animation from the legs.

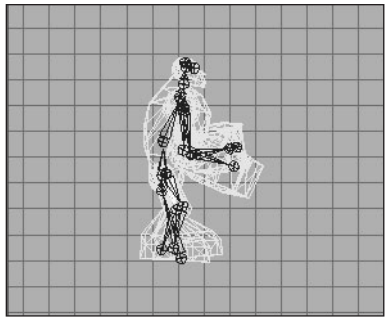


Figure 14.87 Frame 60.

Don't try too hard to make the animation look natural though—he's a goofy character and *should* have that sort of goofy, cartoonlike appearance when running.

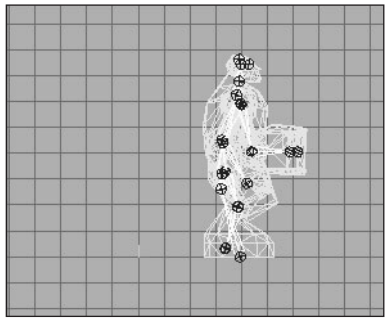


Figure 14.88 Frame 70.

Head Animation

This is the animation that Torque automatically invokes when it needs to know how far your character is looking up or down. So basically this animation's purpose is to define limits or a boundary and not so much the movement. However, if your character's facial or head shape would change when looking up or down, then you would create a more complex head animation.

Team LRN

That being said, it is really quite quickly dealt with.

1. Move to frame 71.
2. In the Right Side view, rotate the head joint until the head is looking up at the maximum angle you want to allow. You may also need to move the head back a bit.
3. Make this a keyframe.
4. Move to frame 72.
5. Rotate the head joint until the head is pointed down at the maximum angle you want to allow. You may also need to move the head forward a bit.
6. Make this a keyframe also.
7. Save your work! There, you are done. That's the entire animation sequence! Check your frames against Figure 14.89 to make sure you got it all right.

Headside Animation

In the same way that the head animation defines the limits for the up and down motion performed by Torque, the headside animation provides the limits for the left and right motion. This is most visible from the third-person perspective when in the game.

Do the same thing you did for the head animation, but use frame 73 for the left turn and frame 74 for the right turn. Make each of these frames a keyframe, and save your work when you finish.

Look Animation

The look animation is basically another movement-limiting animation that defines how the character's arms will be posed when he is looking up or down. Again, it is a simple two-frame animation that doesn't require us to go into in detail now. Use frame 75 for the down "look," or aim. Make sure you have both arms positioned sensibly. Use frame 76 for the up aim. Set both as keyframes and save your work again.

Death Animation

As you saw earlier, there are many possible ways to die. Torque supports 11 "standard" death animations, but you can easily add more by writing a minor code change into the scripts.

We'll cover only one death animation here. We'll have the character collapse backward and fall to the ground on his back with his feet tossing into the air and back down again.

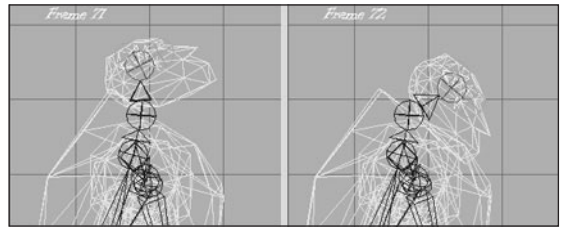


Figure 14.89 Head sequence frames.

1. Move to frame 81 and set the pose back to resemble the resting pose as closely as you can, without spending too much time on it.
2. Set this frame to be a keyframe.
3. Move to frame 90 and rotate the arms and hands to match. You can have the character's head pop off temporarily, like I do, or leave it on but thrown back. It's your model! Let Figure 14.90 guide you.
4. Set frame 90 to be a keyframe.
5. Move to frame 100.
6. In the Side view, drag the base node backward several grid squares.
7. Continue to rotate and move the arms and legs, and rotate the body around the pelvis node to make the body tipped past the horizontal with the bottom of the torso higher than the top, as shown in Figure 14.91.
8. If you haven't guessed it by now, make this frame a keyframe!
9. Move to frame 110.

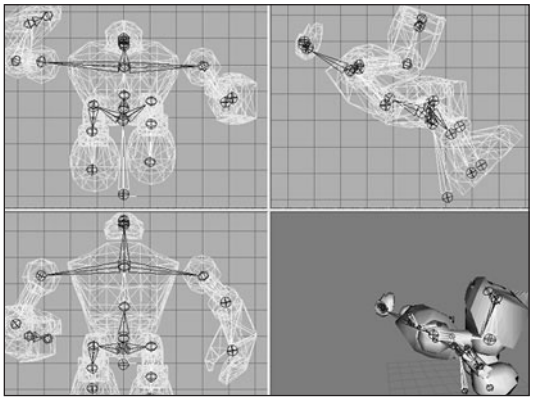


Figure 14.90 Frame 90.

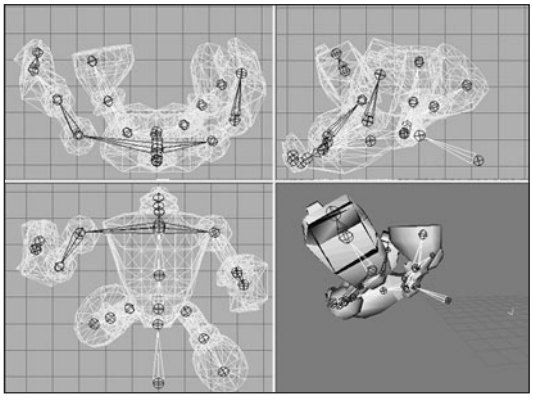


Figure 14.91 Frame 100.

10. The body is hitting the ground, with some momentum still in the legs. Align the bottom of the torso (which is actually the character's back) even with the ground. Rotate the legs and knees to fling the feet up over the body, and rotate the arms to fling them away from the body, as shown in Figure 14.92. By now the base node should be eight or nine grid squares behind the origin along the Z-axis, as seen in the Side view.
11. Yup, this is another keyframe. Go ahead, make its day.
12. Now for the final resting position. Move to frame 120.
13. Lay the body out, flat against the ground. Also, move the base node one or two more grid squares farther back, to cause the body to slide along the ground. Lay the arms flat to the sides, the feet and legs down on the ground and spread somewhat. Tilt the head back. As you can see in Figure 14.93, he's dead, Jim.

Team LRN

14. Keyframe him, Dano! (Okay, that's an obscure reference, I'll admit. Indulge me!)
15. Save your work!

Well, that's the lot of them. Enough animations to give you what you really need to know to get moving on animating for Torque in MilkShape. There's still more to cover—we're not quite out of the woods yet. Now we have to tell Torque how to find the animations.

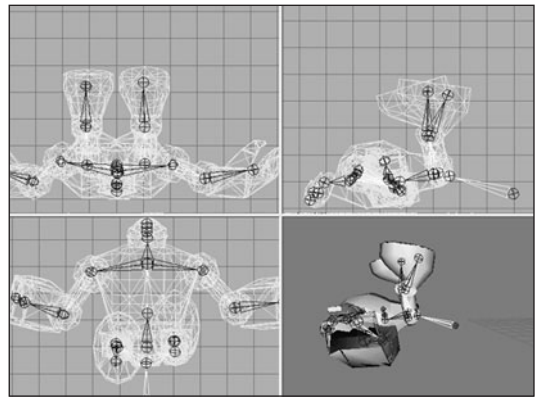


Figure 14.92 Frame 110.

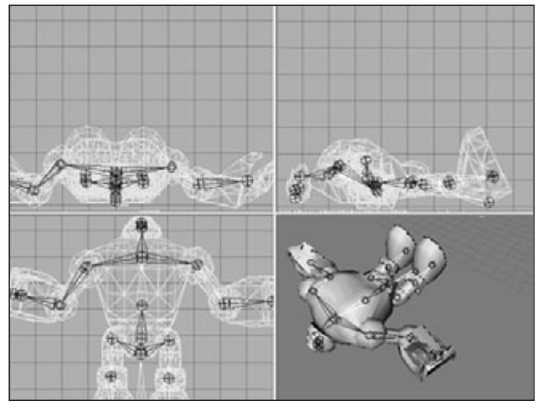


Figure 14.93 Frame 120.

The Animation Sequence Materials

The Torque Engine needs to know where the various animations can be found, how long they run, what type they are, and how fast they should be run. We do this using a technique called the Animation Sequence Materials.

The general approach is that we create a special material, and embedded in the name of that material are the Torque name for the animation sequence, its desired playback frame rate, which frames belong to which sequences (inclusive from start to end), and whether the sequence cycles (loops) or plays once per invocation.

You need to go to the Materials tab of the toolbox and create six new materials—one for each animation sequence. Table 14.3 lists the material names you need to use. The text in the "Sequence Material Name" column must *all* be included in the name, exactly as shown. These special materials tell the Torque DTS Exporter program what special operation to perform on the model as it creates the DTS formatted file for use in Torque.

Note that some of the Sequence Material Names don't include some of the option settings. If you leave them out, the defaults are used. A little later in this chapter you'll find more detail about the exporter.

Finally, there is one more special material we need to make, in order to set the global scale. We built this model on a large scale so we could use the Snap To Grid function without seeing our vertices snapped way out of line. Now, when we export the model, we will need to have it scaled down. Add a material and name it "opt:scale=0.02". This will shrink the model to one-twentieth its created size, which is about right for our needs.

Team LRN

Table 14.3 Animation Sequence Material Names

Torque Sequence Name	Sequence Material Name
root	seq:root=1-30,fps=10,cyclic
run	seq:run=31-70,fps=15,cyclic
head	seq:head=71-72
headside	seq:headside=73-74
look	seq:look=75-76,fps=15
die1	seq:die1=80-120

Exporting the Model for Torque

In the next section of this chapter, we will look at the Torque DTS Exporter for MilkShape, but for now we'll just use it in a fundamental way to get our model to work in Torque.

1. After saving your work, choose File, Export, Torque Game Engine DTS. You will see the Torque Game Engine (DTS) Exporter dialog box appear.
2. We're going to take the defaults, but we should make sure they are correct. You want to have Export animation and Export material information selected, and Collision Mesh should be set to None (Torque handles player collision internally). Click OK when ready.
3. Save your DTS file as C:\3DGPAi1\resources\ch14\myhero.dts.

That was pretty painless. Now let's make sure the model works! We'll dust off the old Torque Show Tool we used in Chapter 9 and check out the model.

1. In Windows Explorer browse your way to C:\3DGPAi1 and launch the Show Book Models shortcut.
2. Click Load Shape.
3. Find RESOURCES/CH14/myhero.dts, choose it, and click Load.
4. The model should appear in the center of the screen, facing away from you.
5. Use the navigation keys (described in Chapter 9 in Table 9.1) to rotate the model and bring it closer to you. You should notice that it is already performing the idle animation (root).
6. Click Thread Control. A window will appear on your screen, probably at the lower right. Drag it to the upper-left corner, or wherever is most convenient.
7. Click Run in the Sequences list. The character should start the run sequence.
8. Check out the other sequences, but remember, the ones that don't cycle are going to run just once and will stay at the last frame. You can use the Transitions button to adjust the transition speeds so that you can check to see if you are getting your desired results.

Team LRN

9. If necessary, go back to your model in MilkShape and make adjustments to your animations, and then come back here to check them out.

Good job! The rule of thumb is, if it works in the Show Tool, it will work in the game, because the Torque Engine is behind both.

You now have an animated Hero character to use in your game! And it really isn't that difficult. If you are even a halfway decent artist and have a good eye, I'm sure your model and animations are much better than mine.

The next section provides some detail into the workings of the DTS Exporter for the Torque Engine. With its help, you should take some time to fiddle with settings and different animations and add your own animation sequences.

The Torque DTS Exporter for MilkShape

The Torque DTS Exporter is included in your C:\3DGPAi1\resources\tools directory and is called ms2dtsexporter.dll. Copy this file into your MilkShape directory, C:\Program Files\MilkShape 3D 1.6.6, and then restart MilkShape. The name of the MilkShape folder changes from version to version, so watch out for that. The exporter shows up under the File, Export menu.

The Torque Game Engine (DTS) Exporter Dialog Box

The Torque Game Engine (DTS) Exporter dialog box (see Figure 14.94) has three groups of options, none of which normally need to be set. Option settings are not saved, so you rarely use this dialog box for more than just a means to double-check your option values. The recommended approach is to set options using special materials.

Collision Mesh

The exporter allows you to create as many collision meshes as you want. Each collision mesh must be named "Collision"; if you have three collision meshes, they will all be named "Collision". If you do not have a collision mesh defined, you may have the exporter create one for you as either a box or a cylinder.

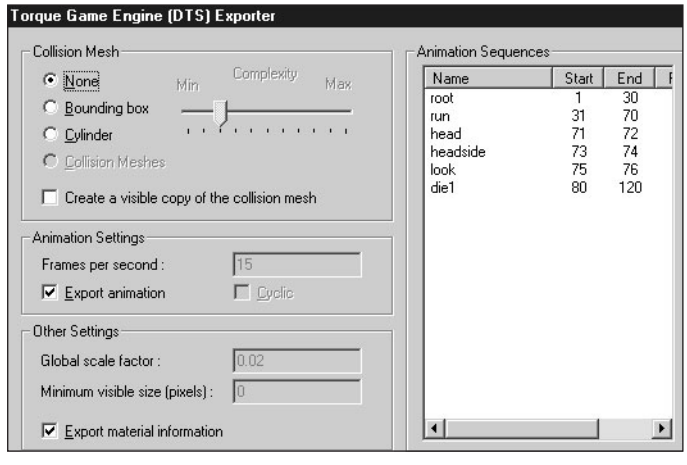


Figure 14.94 Torque Game Engine (DTS) Exporter dialog box.

You can also manually select an existing mesh. Player-characters don't need a collision mesh at all.

Select the Create a visible copy of the collision mesh check box to make the mesh visible as well as collidable.

Animation Settings

The Animation Settings group displays the global values for the Animations.

- **Frames per second.** This field indicates at what speed the Torque Engine should play at the animations. This field can be set using an Export Options material and applies globally to all animation sequences. This does not affect the number of keyframes; it simply sets the rate at which they will be played.
- **Export animation.** If the shape is animated, but no animation sequence has been defined for it, then this field contains the name that will be assigned to it. The sequence will cover the full timeline. This feature is useful for simple models with a single animation sequence.

Other Settings

The Other Settings group contains miscellaneous settings values.

- **Global scale factor.** The global scale factor is the amount by which the shape is scaled when it is exported. The default scale factor is 0.1, but this field can be changed to any value set using an Export Options material.
- **Minimum visible size (pixels).** If the projected screen size of the bounding radius of the shape drops to the minimum visible size, the shape will no longer render. This is normally used to switch between different detail levels, and it's recommended that you leave this at the default value: 0.
- **Export material information.** You may disable the exporting of material information (not recommended) by clearing the Export material information check box.

Special Materials

MilkShape does not directly support a number of Torque Engine features, so the tool has been extended through the use of several clever uses of existing MilkShape features, based on material and mesh names. These are described more fully in the following sections, but they basically fall into two categories: mesh flags embedded in the mesh's name and specially named materials that are used to declare animation sequences and exporter options.

Special materials are materials that are not exported as real materials but are instead used only to store information in the MilkShape file. These materials are processed by the exporter but are not written into the DTS file.

Export Options

Materials with special names can be used to set several export options. These materials are ignored during export and are solely used to set options.

Option materials are named as follows:

```
opt: option, option, ...
```

All other properties of the material are ignored. Table 14.4 lists the available options.

Table 14.4 Export Options

Option	Description
scale=n	The global shape scale factor, where n is a floating point value. The default scale value is 0.1.
size=n	The global minimum visible pixel size. The default is 0.
fps=n	The global default frames per second value for animations. Each animation sequence may set this value, but if it's not defined by the sequence, this default value is used.
cyclic	The global default animation looping flag. Each animation sequence may set this value, but if it's not defined by the sequence, this default value is used.

There may be more than one option material. If the same options are set on multiple materials, then the last one in the material list is the value used. Here are a couple examples of valid material names:

```
opt: fps=10, cyclic
opt: scale=0.1
```

Material Option Flags

Material attributes can be set using the MilkShape Shininess and Translucency sliders as well as by embedding additional flags in the material name.

Environment mapping can be controlled for the model by use of the Shininess slider—it's the one on the left-hand side. Setting the slider to any value but 0 will enable environment mapping for the texture. Note that the texture you are using must have an alpha channel, which will be used to control the per-pixel shininess of the texture. Any value of the slider other than 1.0 or 0.0 will be ignored.

You can enable translucency by setting the MilkShape Translucency slider—this is the slider on the right-hand side. Setting the slider to any value other than 1.0, which is to the far right, will enable translucency for the texture. The texture you are using must have an

alpha channel, which will be used by the Torque Engine to control the per-pixel translucency of the texture. Any value of the slider other than 1.0 or 0.0 will be ignored.

Options that are embedded in the material name follow this format:

```
name: flag, flag, ...
```

where the : and `flags` are optional. Table 14.5 shows which flags are available.

Table 14.5 Material Option Flags

Flag	Description
Add	Enables additive transparency.
Sub	Enables subtractive transparency.
Illum	Enables self-illumination (lighting doesn't affect it).
NoMip	Disables mipmapping.
MipZero	Sets the "MipMapZeroBorder" flag.

A self-illuminating additive material could be called as follows:

```
Flare: Add, Illum
```

Mesh Option Flags

Meshes can have additional flags embedded in the mesh (or group) name. The mesh name follows this format:

```
name: flag, flag, ...
```

where the : and `flags` are optional. Table 14.6 shows which flags are available.

Table 14.6 Mesh Option Flags

Flag	Description
Billboard	The mesh always faces the viewer.
BillboardZ	The mesh faces the viewer but is only rotated around the mesh's Z-axis.
ENormals	This flag encodes vertex normals. It is deprecated and should not be used unless you know what you're doing.

Here are some legal mesh or group names:

- leaf
- leaf: Billboard
- leaf: BillboardZ

By default, meshes do not have any flags set.

Animation Sequences

MilkShape only provides a single animation timeline, but the Torque Engine supports multiple animation sequences, each of which can be named and have different properties. Multiple sequences in MilkShape are animated on the main timeline and are split into separate sequences by the exporter. For this to happen animation sequences must be declared, indicating where each sequence starts and ends on the master timeline. This is done through materials with special names. These materials are ignored during export and are solely used to declare animation sequences. The section on Special Materials above provides more details.

Sequence materials are named as follows:

```
"seq: option, option, …"
```

All other properties of the material are ignored. Table 14.7 describes the Sequence Material Options.

Table 14.7 Sequence Material Options

Option	Description
name=start-end	This declares the name of the sequence followed by the starting and ending keyframes. This option must exist for the sequence declaration to be valid.
fps=n	This is the number of frames per second. This value affects the duration and playback speed of the sequence.
cyclic	Sequences are noncyclic by default. Cyclic animations automatically loop back to the start and never end.

Here are some valid sequence declarations:

```
seq: fire=1-4
seq: rotate=5-8, cyclic, fps=2
seq: reload=9-12, fps=5
```

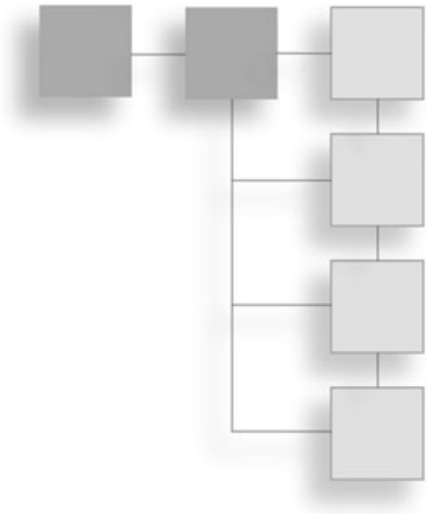
Moving Right Along

That was a pretty busy chapter, huh? We created a character model and a texture skin for it, created a skeleton and rigged the model's meshes to the skeleton, and then proceeded to animate the skeleton. It's a lot of work, and that's why even the smallest game development team usually has at least one modeler on board to handle that workload.

So now that you can create your own player-character, it's time to create some sort of transportation so he can get around in the game world. That's the subject of the next chapter, "Making a Vehicle Model."

Team LRN

CHAPTER 15



Making a Vehicle Model

In Chapter 9 we looked at creating skins, and during that process we created a skin for a cool runabout-type vehicle. In this chapter we'll create the model itself.

Of course, there is a whole host of different vehicle types, such as those for the open road or off road, aircraft, hovercraft, ships, and so on. In most cases the methods used to create the vehicles can be used with any type of object. But different vehicle types have different capabilities and therefore may require some specialized accommodation for those capabilities in the models.

For example, if you want your vehicles to be drivable by a player-avatar that gets into the vehicle when you want to drive, then your vehicle will need *mount points*, which are special nodes or subobjects within the model that indicate where your player-avatar gets attached to the vehicle. Different vehicle types, makes, and models will have different needs in this area.

Then there are those special nodes that indicate where other game functions will happen: Wheeled vehicles need to know where the wheels are located, as well as information about how the springs and steering mechanisms are oriented.

Some vehicles might need nodes in their models to indicate to the engine where to generate engine exhaust smoke using particles. Flying vehicles may require nodes to help the engine generate contrails (condensation trails). The list goes on.

Team LRN

The Vehicle Model

In this chapter we are going to build a complete wheeled vehicle—the runabout, which will wear the skin you created in Chapter 9. Then we will insert it into a little test game so that we can carom about and drive our insurance rates through the roof!

The Sketch

I find the best way to start a new model is with a sketch. Doodle out some ideas, and keep working them up on paper until you get something that suits your needs. Then choose a view (Left or Right, Top or Bottom, Front or Back) that presents you with the most number of intersections of lines to use as points.

Figure 15.1 shows a sketch of the runabout from the right-hand side. Notice that it is really a sketch and not a drawing. As long as the general proportions and coarse features are present, it's satisfactory. Now if you were going to model a real car, you might need to use a more detailed sketch or perhaps something that would qualify more as a drawing than a sketch. Then again you may not—it all depends on how much detail is necessary to suit your needs.

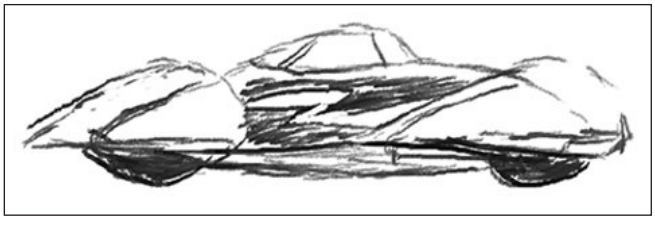


Figure 15.1 Side view sketch of the runabout.

The Side view is the main sketch we will use to make our model. When modeling most vehicles, you will *usually* use a Side view as your primary source for extrusion modeling. The reason for this is pretty obvious: Most vehicles are longer than they are tall or wide. The symmetries of vehicles (how one side mirrors the other) tend to reflect around the longitudinal axis, the one that runs from the rear to the front of the vehicle up the middle.

Figure 15.2 Top view sketch of the runabout.

The Top view of the runabout is shown in Figure 15.2. The purpose for sketching this view is to provide a guide for your modeling efforts as they proceed. You will find yourself checking back against this drawing quite often.

One more useful thing is to make a copy of the Side view and, using Paint Shop Pro, adjust the brightness of the image. This is because when we import the sketch into MilkShape, we don't want the image to overpower any of the on-screen modeling marks we make. I find the best approach is to darken the whole image by around 40 to 50 percent and reduce the contrast by about 50 to 60 percent or so. Figure 15.3 shows the

Team LRN

adjusted Side view. I keep the original sketches as they were so that I can print them out for reference purposes (and also just to pin up on the wall because it's a cool artsy thing to do).

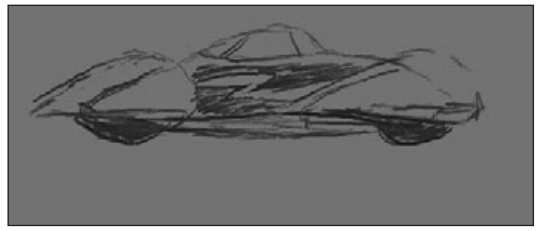


Figure 15.3 Side view sketch adjusted for use in MilkShape.

The Model

So pour some gasoline in the ol' computer, grab the pull-start, and fire up MilkShape again if it isn't already running. If you need a quick refresher, you can jump back to Chapter 14. Set your MilkShape GUI display to the four-view mode by choosing Window, Viewports, 4 Window.

Create a fresh new MilkShape document. It's a good idea to save the empty file right now, just to get the path set up and establish the file name.

Then you want to import the side sketch as the background in the Side view (Top Right view). Right-click in the Side view and choose Choose Background Image from the pop-up menu. Select your sketch and click OK. You can use my sketch if you want—it's located at C:\3DGPAi1\RESOURCES\CH15\ref_sketch.bmp.

You should end up with something like Figure 15.4.

So now we start.

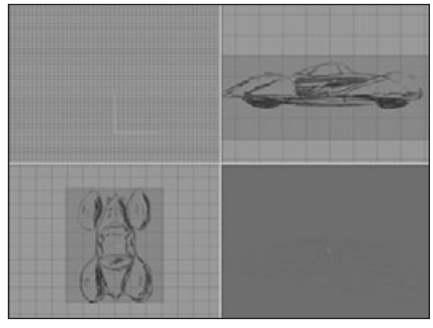


Figure 15.4 MilkShape windows with reference sketch.

Building the Body

First, we'll build the body:

1. Select the Vertex tool, making sure that the Auto Tool check box is cleared.
2. In the Side view, start placing vertices at all the major corners and points around the edge of the car's body—don't do the fenders yet. See Figure 15.5 for reference.

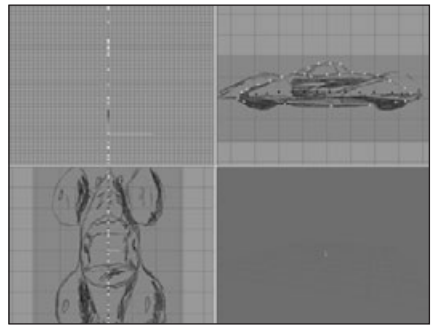


Figure 15.5 Placing vertices over the reference sketch.

note

There is also a string of vertices across the "waist" of the body—in Figure 15.5 they are highlighted as black squares (they show red in MilkShape). These extra vertices are added for two reasons: First, they act as useful anchors for creating the faces that we'll make later. And second, they help add more malleability to the model for shaping the sides of the car.

After we have the vertices placed, we move on to creating the faces joined by the vertices.

tip

It's important to remember that when we create faces using the Face tool, we click on three vertices in sequence to create one face. The order we click on the vertices is important. To create a face or polygon that is facing toward us, we need to select the vertices in a counterclockwise order, as shown in Figure 15.6. For most cases this is not hard to do, but it's possible to get confused and lose track of the sequence. In this case you can use the Edit, Undo menu item to back up until the sequence is clear. You can also abort any three-vertex sequence by just clicking on the Selection tool (or any other tool) and then clicking back on the Face tool again. Then you can start with a fresh trio of vertices.

3. Starting at the right side (the front of the car), begin creating faces, moving to the left along the top as you proceed, including the window area, as shown in Figure 15.7.

   When you reach the left side, you should have something resembling Figure 15.8.
4. After completing the top row of faces, start making faces along the bottom, from the left back over to the right (see Figure 15.9).
5. Finish up the faces for the side of the body. You should end up with something like Figure 15.10.

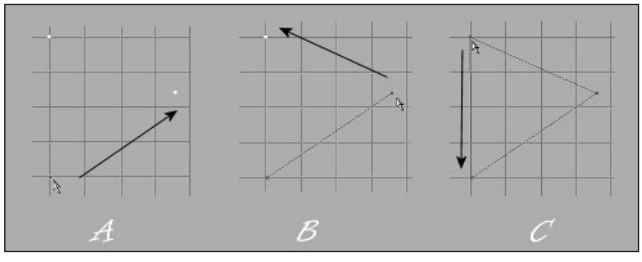


Figure 15.6 Vertex order for creating faces.

Okay, so now we have a plane of body faces. We want to make sure they are all oriented correctly. The quickest way to do this is to look at the output in the 3D Perspective view. Make sure the view (it should be the Bottom Right view) is set to

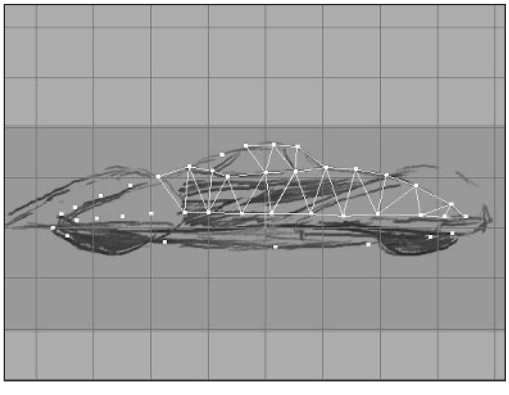


Figure 15.7 Creating faces starting from the right.

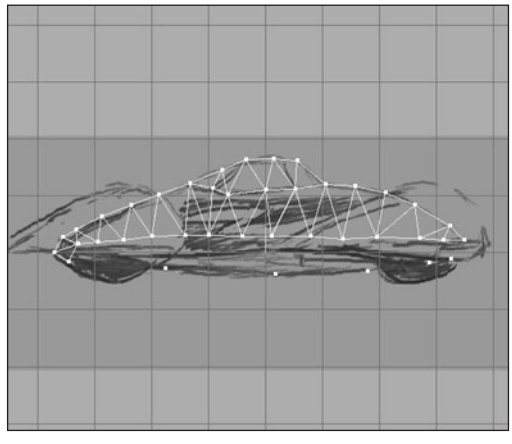


Figure 15.8 Finishing the top row of faces.

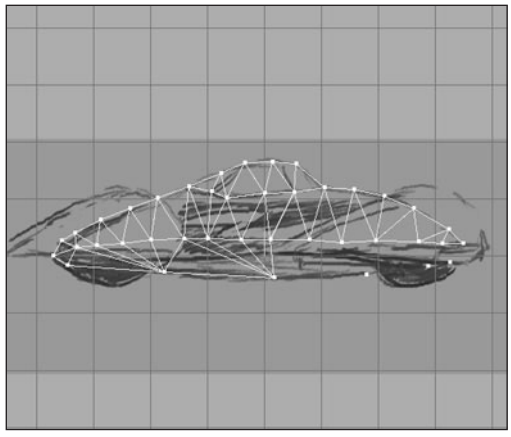


Figure 15.9 Working the bottom row of faces.

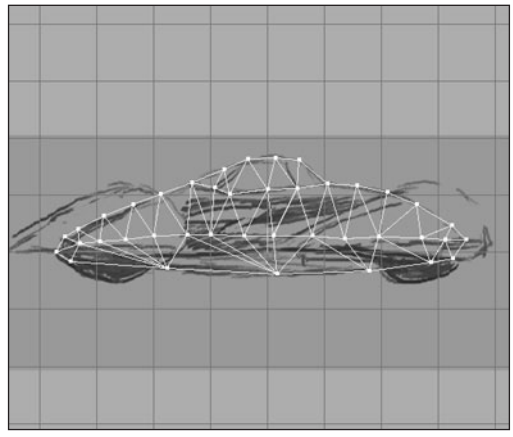


Figure 15.10 Completed plane of body faces.

either Flat Shaded or Smooth Shaded by right-clicking on the view and then choosing either Flat Shaded or Smooth Shaded in the pop-up menu.

What you should see is the outline of the body rendered in white or light gray, just as in Figure 15.11.

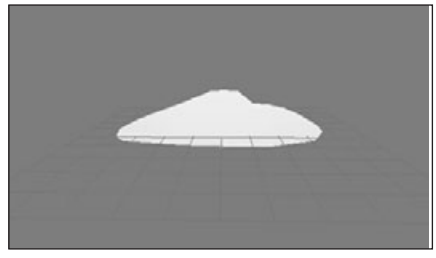


Figure 15.11 The 3D view of the initial body faces.

tip

This is a long one, so make sure you've got some popcorn handy!

Sometimes you end up with two overlapping faces: one oriented correctly, and the other reversed. These are hard to catch until they start showing strange results when rendering, as the model grows more complex. There is another way to check for misoriented faces that is a little more involved:

1. Pick the Selection tool and set it to Face mode. Make sure that the Ignore Backfaces check box is *selected*.
2. Now use the Selection tool to select all the faces by dragging the selection rectangle around all of them. This will highlight all the faces.
3. Now choose Edit, Hide Selection. All of the correctly oriented faces will vanish, leaving behind only the ones facing the wrong way.
4. To fix the problem, unhide all the hidden faces, *clear* the Ignore Backfaces check box, and select all the faces again. This will select all correct and incorrect faces.
5. Then choose Face, Reverse Vertex Order. This will make the good ones bad, and the bad ones good. Still with me?
6. Okay, now deselect everything by clicking the Select tool in an open area, and *select* the Ignore Backfaces check box again.

Team LRN

7. Drag select over all the faces one more time. Now only the faces that were originally incorrect will be selected.
8. Choose Face, Reverse Vertex Order with those faces selected.
9. Then for one final time *clear* the Ignore Backfaces check box, select all the faces, and choose Reverse Vertex Order. This should flip all faces back to the correct orientation. If this reminds you of manipulating a Rubik's Cube, then you think a lot like I do!

So now we'll move on to adding some width to the body.

6. Choose the Selection tool and set it to Face mode.
7. Select all the faces.
8. Click the Extrude tool and fill in the X entry of the XYZ boxes with the value −10.0 (that's minus ten point zero). Leave Y and Z at 0.0.

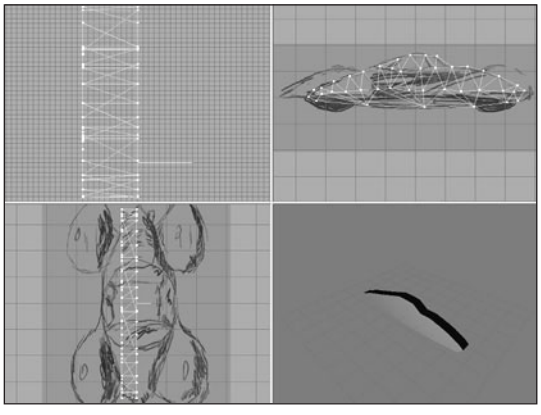


Figure 15.12 First extrusion.

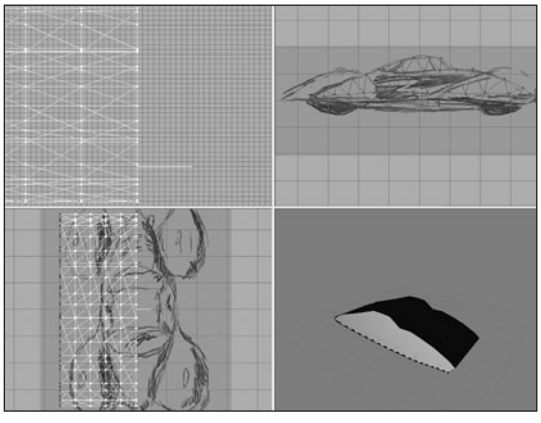


Figure 15.13 After extruding the body faces five times.

9. Click on the Extrude button to the right of the XYZ boxes. You should get a new set of polygons negatively offset in the X-axis by 10 units, as shown in Figure 15.12.

   Notice the way that the image in the 3D view looks—now the body has some depth to it. It's not just a plane of faces anymore.
10. Repeat step 9 four more times, until you get five segments as shown in Figure 15.13. Warning: Do not click on any other tool or in the edit windows. After the fifth extrusion you want to end up with the body faces still selected.
11. The body faces should still be selected if you got my warning in time. Choose Edit, Duplicate Selection.
12. Click the Move tool, then go to the Top view at the lower left, and drag the highlighted faces (these will now be the copies, not the originals) to the right, clear of the extrusion segments, so that you get something like that shown in Figure 15.14.

13. Choose Face, Reverse Vertex Order.
14. Using the Side views and Top views, align the vertices of the copy of the body faces with their counterparts in the main body.
15. In the Top view drag the body face copy over to the right edge of the rest of the body polygons. Align the vertices as best you can by eye.

tip

We want to make sure that the vertices in Step 15 are perfectly aligned. To do this, we'll scale our entire model up. Making our model larger in relation to the grid allows us more precision with the grid. This will help ensure good results when we snap our vertices to the grid, which is going to happen shortly.

16. Select the entire set of polygons in all the faces, and then use the Scale tool to make the entire model four times larger.
17. Select the entire model in vertex selection mode, and choose Vertex, Snap To Grid.
18. Choose Vertex, Weld Vertices.
19. Scale the model by 0.25. This will restore the model back to its original size.
20. In the Top view make sure the entire model is selected, and then use the Move tool to drag the model over the sketch so that it is aligned around the longitudinal center of the car in the sketch.

    You should now have a model that looks like that shown in Figure 15.15.
21. In the Top view select the bottom nine rows (or forwardmost nine rows) of vertices.
22. Use the Scale tool to scale the selection to 0.9 in the X-axis only; leave the other values at 0.0. Figure 15.16 shows the result of this operation.
23. Change your selection to be the bottom eight rows and scale to 0.9.

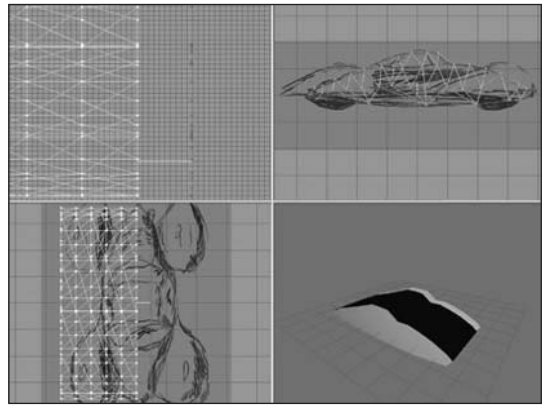


Figure 15.14 After duplicating and moving the copies.

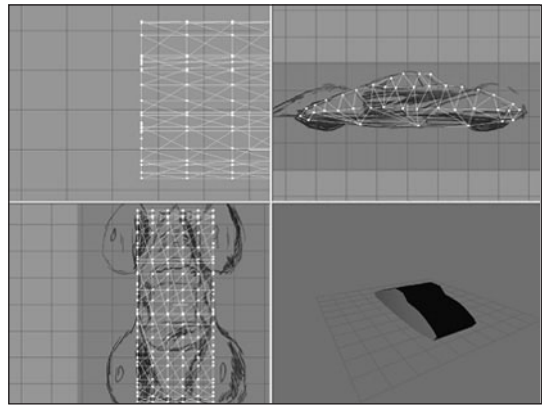


Figure 15.15 Scaling the nose of the runabout.

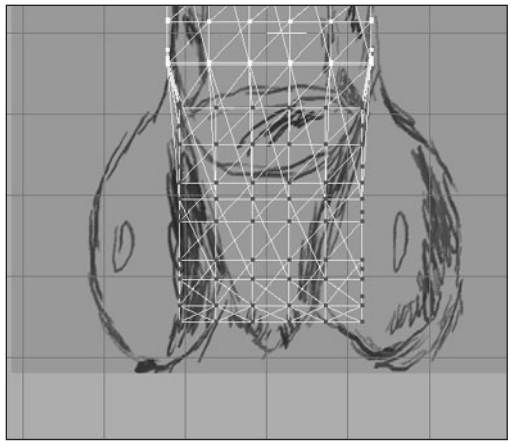


Figure 15.16 After scaling the bottom nine rows.

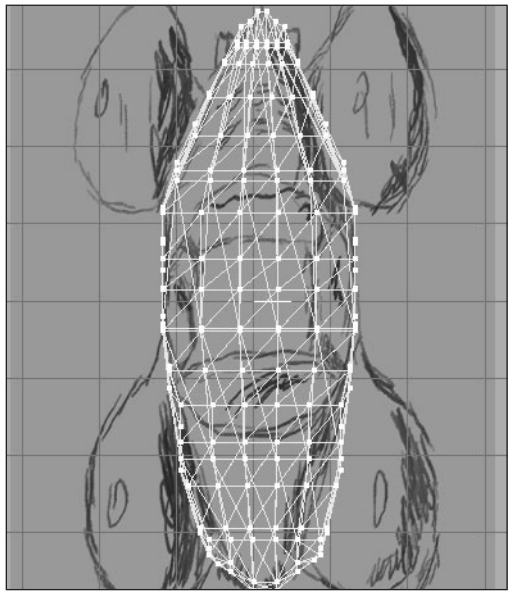


Figure 15.18 After scaling the tail.

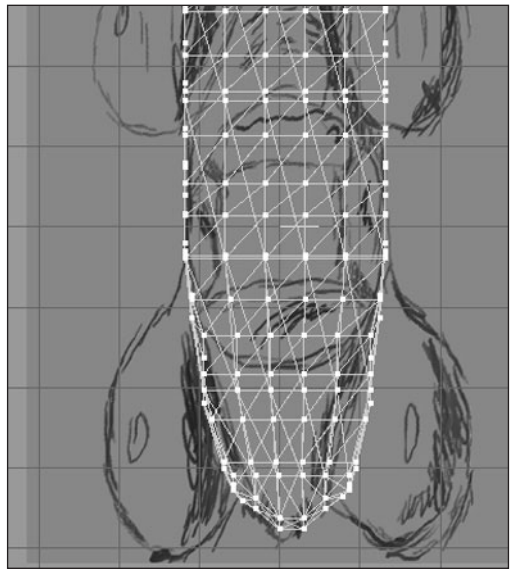


Figure 15.17 After scaling the nose.

24. Repeat the decrementing and scaling of the selection, reducing your selection vertices one row at a time until you run out of victims…ummm…I mean vertices.

    You should now have something that resembles that shown in Figure 15.17.

25. Repeat this iterative scaling process for the rows of vertices as seen in the Top view at the other end of the car body, until it, too, tapers, as shown in Figure 15.18. You may find it necessary to manually move a few vertices at either end to achieve the appropriate amount of taper.
26. Now perform the same sort of iterative scaling operations on the Front view of the car, getting it to look something like the view shown in Figure 15.19.
27. Next, use the Selection and Move tools to place the car so that it is centered around the origin (0,0,0), as shown in Figure 15.20. The axis "bug" at the origin has been enhanced as thick black lines to emphasize its location.
28. Finally, select all the polygons and use the Groups tools to regroup all polygons into a single group—name it "body".

Team LRN

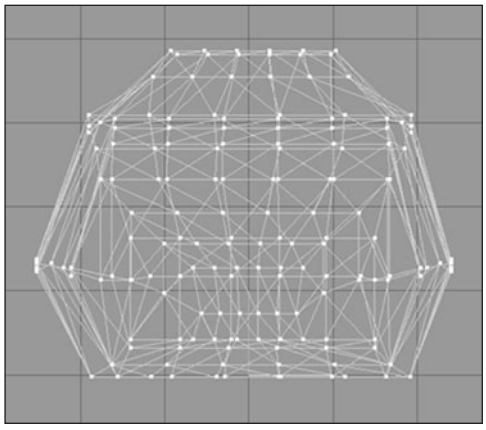


Figure 15.19 Shaping the Front view.

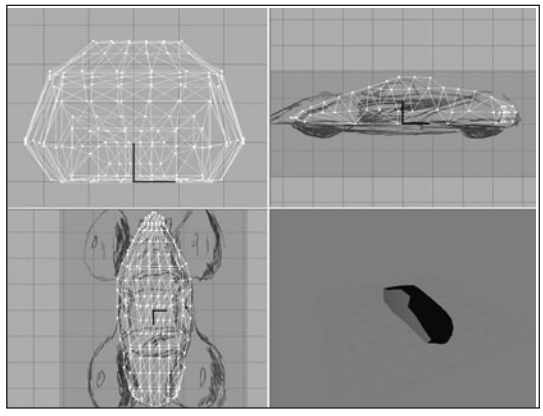


Figure 15.20 Centering the Front view.

Building the Fenders

Next, we will tackle the wheel well and fender assemblies.

1. Hide the body group.
2. On the Model tab, select the Sphere tool, and create a sphere that matches the forward curves of the forward fender, as shown in Figure 15.21.
3. Select the bottom two rows of faces and delete them. Then move the bottom row of vertices up a bit, to get something that looks like Figure 15.22.
4. Select the leftmost three rows of vertices and move them farther left, as shown in Figure 15.23.
5. Continue to reshape the fender to match the sketch as shown in Figure 15.24, until you are happy.

   The next bit is a little tricky, so move slowly. We want to drag certain of the vertices from the fender over to the exact position of vertices on the body. The vertex rows

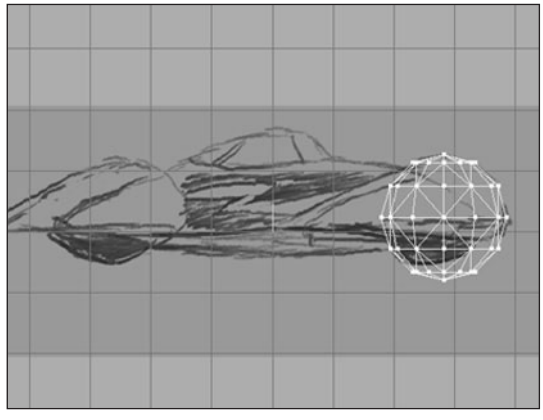


Figure 15.21 Fender sphere.

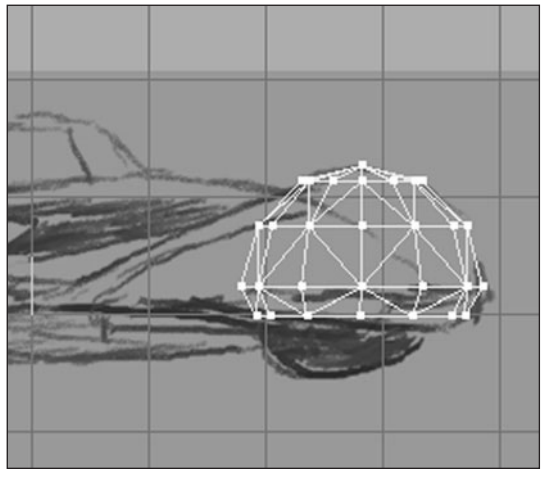


Figure 15.22 Lopping off the bottom of the fender sphere.

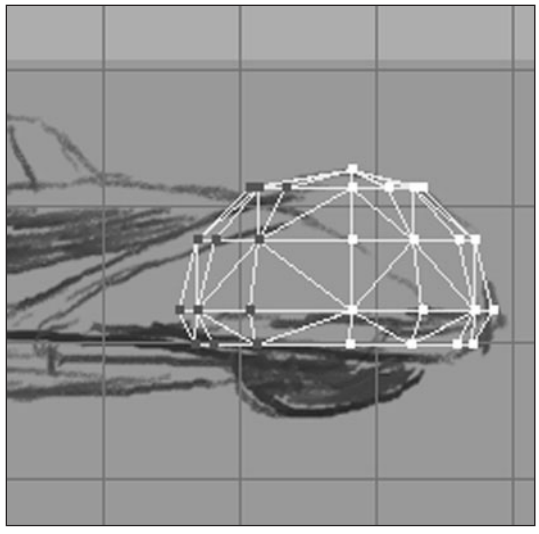


Figure 15.23 Stretching the fender.

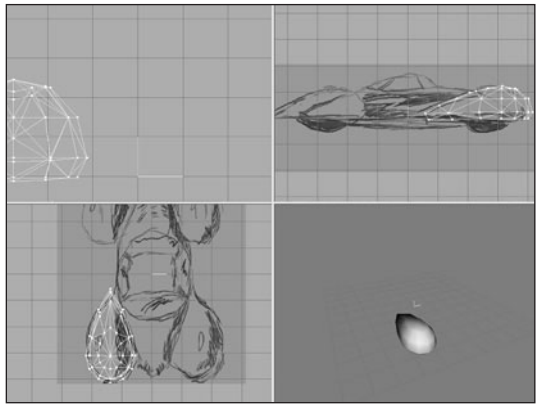


Figure 15.24 Shaping the fender.

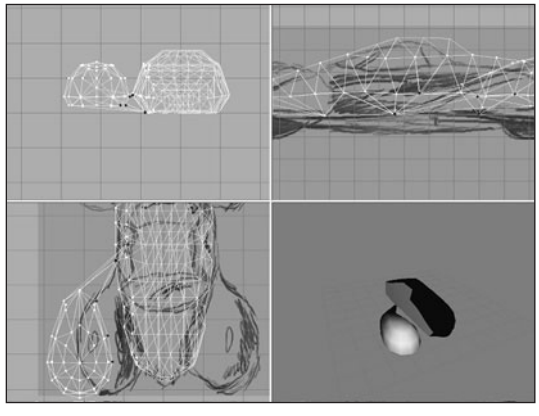


Figure 15.25 The fender vertices.

we want from the fender are the two bottom ones, and we are interested in the vertices on the body side. By dragging them over to the body, we create a fairing-cum-running board sort of affair.

6. Unhide the body.
7. Drag the fender away from the body so that it is in the clear in the Top and Front views.
8. Select the vertices and drag them, one at a time, as shown in Figure 15.25. The vertices are on the two bottom rows and are the ones that face the body.

   Make sure to place the vertices exactly where they mate with a corresponding vertex. A close-up view is shown in Figure 15.26.
9. After each vertex is placed with a mate, use Snap To Grid to make sure they are exactly coincident, and use Weld Vertex to convert each pair of vertices into one vertex. Once you have done this for all the appropriate vertices along the fender bottom, you will get something like Figure 15.27.
10. Finally, move the fender back to a position that matches the sketched fender in the Top view, as shown in Figure 15.28.
11. Repeat steps 2 to 10 for each of the other fenders. Remember to hide the body and the other fenders when necessary to remove clutter from the screen. You should end up with the finished car as shown in Figure 15.29. But we aren't done yet!

The Mount Nodes

In Chapter 14 you learned how to make a skeleton for an animated character in MilkShape using joints. In this section we are going to use the same feature, the joint, to create *nodes* that tell Torque where to mount certain things on models.

1. As shown in Figure 15.30, create four *unconnected* joints, or mounts, on the four corners of the car where the wheel hubs would be. To ensure that the joints are unconnected, you need to use the Select tool to deselect each node after it's been created.
2. Name each of the joints with the names shown in Figure 15.30, with hub0 being the left front joint.
3. Add two more unconnected joints to the locations shown in Figure 15.31. Name the front one "eye" and the rear one "cam".
4. Finally, add two unconnected joints to the locations shown in Figure 15.32. Name the one on the right (the left-hand seat position) "mount0" and the other one "mount1".

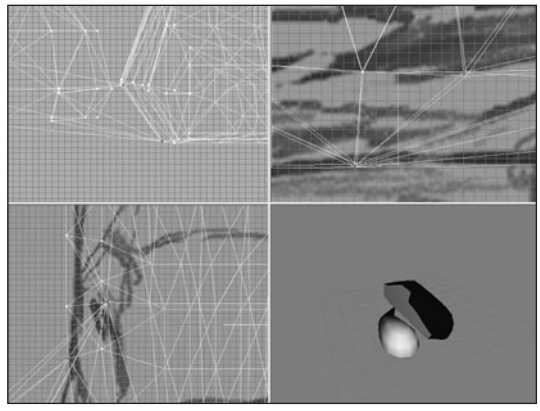


Figure 15.26 Close-up of moved vertices.

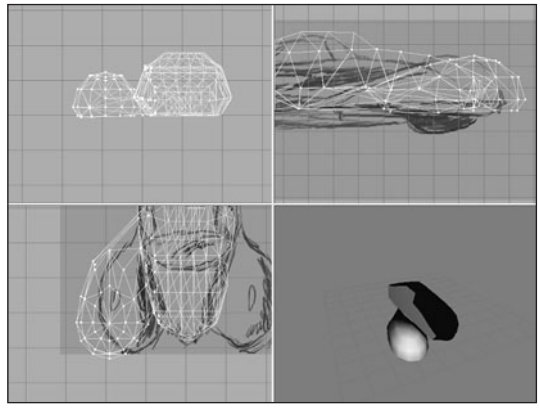


Figure 15.27 All vertices moved.

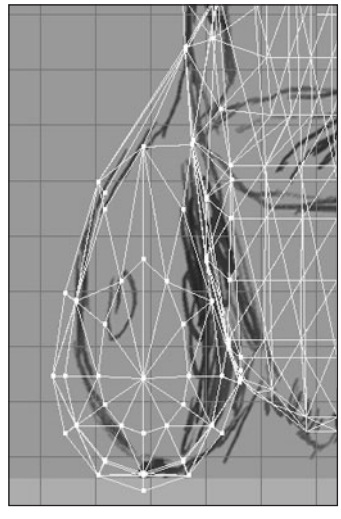


Figure 15.28 Finished fender.

Now, the last two pairs of mounts are used for different, and mutually exclusive, purposes. The eye and cam mounts are used for games where the car becomes the player's avatar. The sample racing game that comes with Torque works like that. The eye node located at the point of the eye mount is the normal first-person point-of-view location for the view's eye. The cam node is for the third-person point of view—the actual camera is offset from the location of this node and so is usually actually behind and above the vehicle.

The mount0 and mount1 mounts are used for games where the player's character actually gets "in" the vehicle;

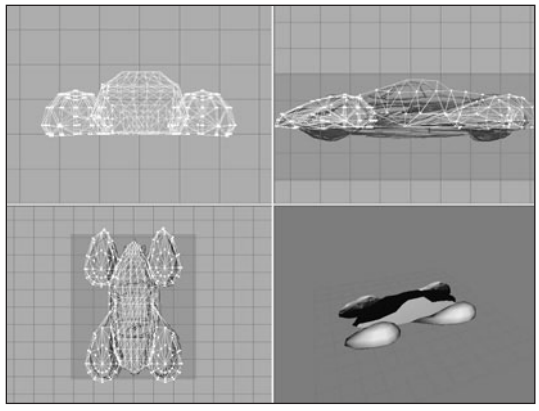


Figure 15.29 All fenders and body completed.

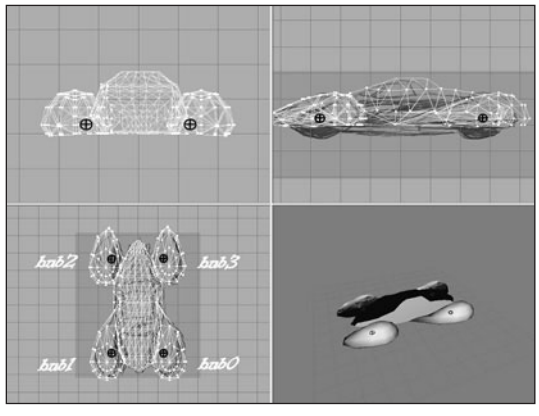


Figure 15.30 Mounts on all four corners.

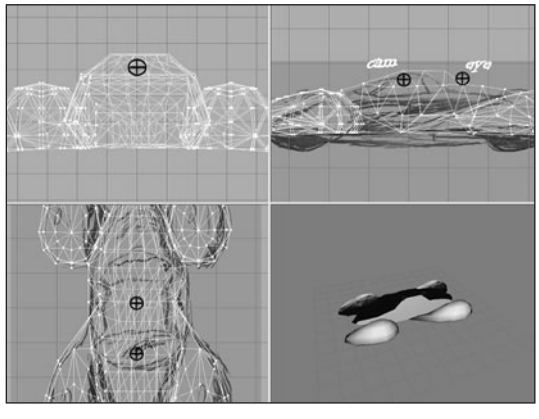


Figure 15.31 Eye and camera mounts.

they specify where the player's avatar will be mounted. The game continues to use the player's avatar's camera and eye nodes. You saw those in use back in Chapter 14.

Skins

Chapter 9 covered the subject of skins and UV mapping, so I refer you back there to map the textures for your new car. You can find a copy of the skin to use at C:\3DGPAi1\RESOURCES\CH15\runabout.jpg.

Collision Mesh

For all objects except player models, we need to create at least one collision mesh if we want the engine to detect when it collides with another object, so use the Box tool in the Model tab to create a box that surrounds the vehicle, as shown in Figure 15.33.

Name the collision mesh "Collision". Any name that starts with "collision" will do, like "Collision-1", "CollisionA", and so on. You can have more than one collision mesh if you like.

You should also hide the collision mesh and then save the model before exporting the model.

The Wheels

Of course, a cool car needs cool wheels. There's not much to them, so I invite you to model your own wheel for use on the car. You, of course, may decide to make a complex model, but there is really no need—a lot can be done with a decent skin. A collision mesh is not needed, but do make sure that the wheel is oriented in the views the same as it would appear if attached to the car. Also ensure that the axle of the wheel and

the midline of the tire are aligned with the origin bug.

Testing Your Runabout

In order to test the runabout, we first need to export it from MilkShape.

1. After saving your work, choose File, Export, Torque Game Engine DTS. You will see the Torque Game Engine (DTS) Exporter dialog box appear.
2. Use defaults, but make sure they are correct.

   You want to have Export animation and Export material information selected, and Collision Mesh should be set to Collision Meshes (this is automatic if the exporter finds a mesh whose name starts with "Collision"). Click OK when ready.
3. Export your runabout to DTS format as C:\3DGPAi1\racing\data\shapes\car\runabout.dts.
4. Open your wheel model and export it as C:\3DGPAi1\racing\data\shapes\car\wheel.dts.

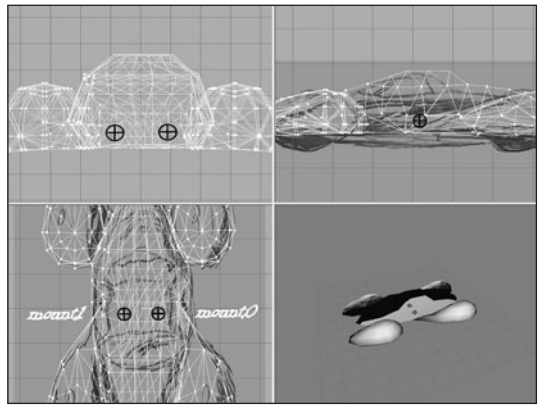


Figure 15.32 Seat mounts.

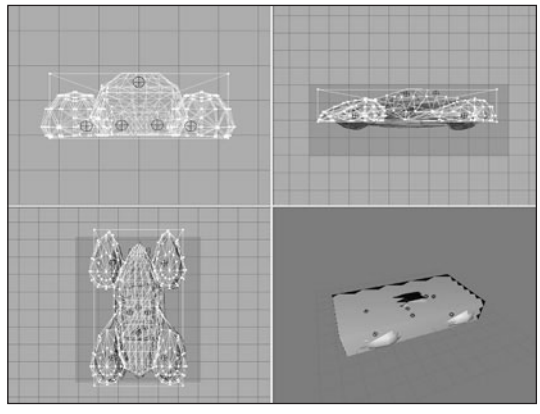


Figure 15.33 Collision mesh.

Next, you need to edit the script that controls the vehicle so it will look for your model and not the default one.

1. Locate the file C:\3DGPAi1\racing\server\scripts\car.cs and open it with UltraEdit.
2. There are two lines that say this:

   ```
   shapeFile = "~/data/shapes/rifle/weapon.dts";
   ```

   Replace each of them with this line:

   ```
   shapeFile = "~/data/shapes/tommygun/tommygun.dts";
   ```
3. Then find the line that says this:

   ```
   shapeFile = "~/data/shapes/buggy/wheel.dts";
   ```

And replace it with this line:

```
shapeFile = "~/data/shapes/car/wheel.dts";
```

4. Save the file.

Okay, now it's time to run the racing demo. The following may look familiar to you, since we did this back in Chapter 9.

1. Browse to C:\3DGPAi1 and click the Run racing Demo shortcut.
2. Click Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Car Race Track from the mission list.
5. Click Launch.
6. After the game loads, have at it! You probably should switch to Chase view by pressing the Tab key—there's more to see. See Table 15.1 for the keyboard controls.

Table 15.1 Torque Racing Demo Controls

Key	Description
mouse	steering left or right
Up Arrow	accelerate
Down Arrow	brake
Tab	toggle from first- to third-person viewpoint
Escape	exit the game

Moving Right Along

Building the model is only half the battle—well, maybe three-quarters. There is still the matter of defining the vehicle's characteristics, like mass, drag, speeds, particle generators, collision handlers, and so on. You take care of these things in scripts that run on the server.

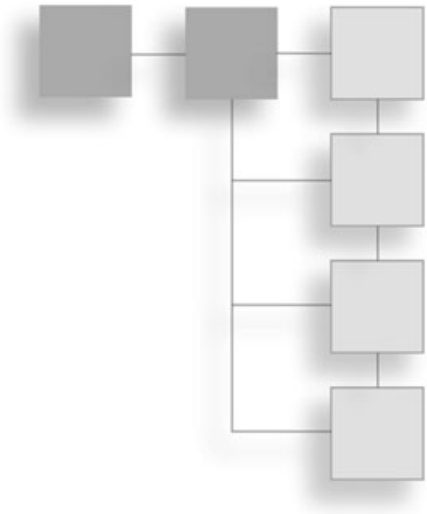
To do the testing you just did, you used the existing demo buggy script that comes with the Torque Engine and simply substituted our model in place of the dune buggy. It looks like a roadster but drives like a dune buggy! In fact you will recall that you had a test-drive of a dune buggy back in Chapter 9.

Later, in Chapter 22, we will create the script that will define the behavior of the vehicle that we've modeled here and its response to user inputs and game environment stimuli.

Coming up next in Chapter 16, we'll continue with MilkShape and make some weapons and other items.

CHAPTER 16

Making Weapons and Items



In this chapter we're going to make a bunch of things. Most of the techniques used will basically be a review for you, so you can see this chapter as one big exercise in applying what you've learned to different situations.

We're going to make a few weapons, and in order to maintain balance, we'll make something that can be used in game to counteract the effects of these weapons.

We'll also make some items that one might call *decorations* for the game. The purpose of these items—some trees and a rock—is to provide some clutter. This is to help fill out otherwise sterile-looking game worlds, making them more interesting to wander around in.

The Health Kit

We'll start out with an easy one. Like I said, this will be a basic review, but it's important to go over the process involved in creating an item for use in the game so that the broad steps become obvious and second nature.

The Model

The Health Kit is little more than a fancy-looking box, as you can see from Figure 16.1. So this won't take long.

1. Fire up MilkShape and create an empty document.
2. Use the Box tool to create a box, as shown in Figure 16.2.
3. Align the box to be centered at the origin for all three axes, as you can see in Figure 16.2.

Team LRN

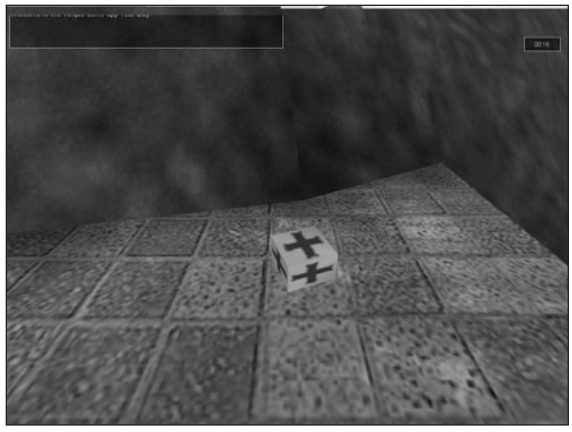


Figure 16.1 The Health Kit in game.

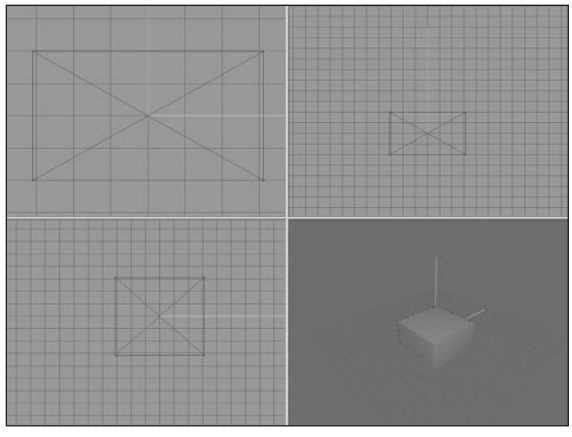


Figure 16.2 The box.

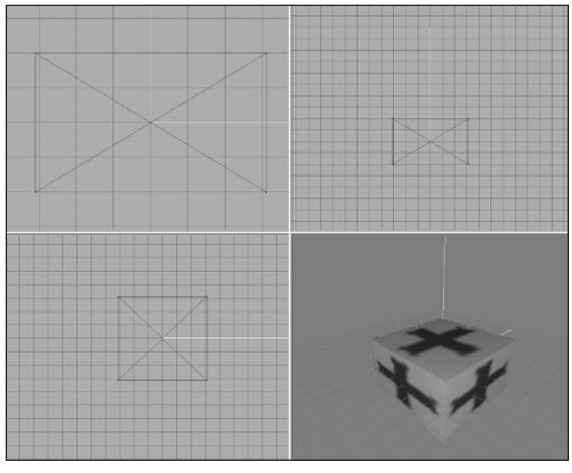


Figure 16.3 The Health Kit model.

4. On the Materials tab, create a new material, using C:\3DGPAi1\RESOURCES\CH16 \ healthkit.png as the bitmap.
5. Name the material "healthkit".
6. Select the box and assign the healthkit material to it.
7. Make sure that the 3D view has been set to texture mode. You should see a nice, shiny new first-aid kit kind of item there, like that in Figure 16.3. This one always has bandages in it!
8. Save your work.
9. After saving your work, choose File, Export, Torque Game Engine DTS. You will see the Torque Game Engine (DTS) Exporter appear.
10. You want to take the defaults but have Collision Mesh set to None.
11. Export the box to C:\3DGPAi1\ fps\data\shapes\items\healthk-its.dts. Click Yes if you get an alert asking if you want to replace an existing file with that name.

Testing the Health Kit

To use the Health Kit in game, you merely have to run over it to pick it up. Then you activate it by pressing the "h" key to restore your health whenever it gets too low. You may remember using a kind of first-aid kit in one of your sample games from an earlier chapter—Emaga5—where you got health back just by running over the first-aid kit, or Health Kit. This one you have to

pick up and activate; we'll test that functionality later when we get back into server scripts. Right now we just want to see our fine creation in the game world.

1. Browse to C:\3DGPAi1 and click the Run fps Demo shortcut.
2. Click Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Water World from the mission list.
5. Click Launch.
6. After the game loads, look around for a big gray block jutting out from the side of a hill into the water. Figure 16.4 shows what the block (marked by the arrow) looks like.
7. Run over to the block. On top of it you will find the Health Kit. You can stand there and admire it for a while if you like.
8. Run over the kit to automatically pick it up. You will see a message in your chat window telling you that you've picked it up.

A Rock

Oh, big deal, a rock—what's up with that, you ask? Well, it *is* going to be your own handmade rock! That should be worth something.

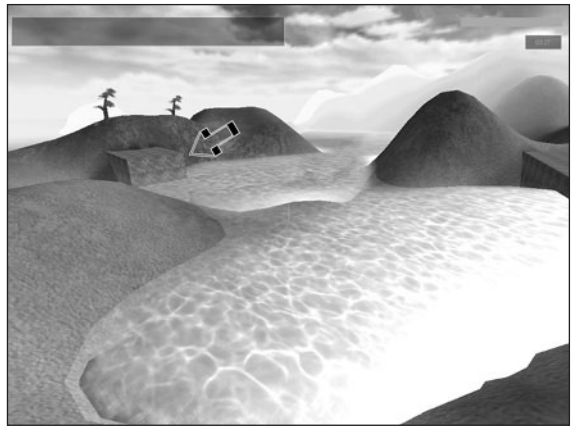


Figure 16.4 Locating the big gray block.

The point here is that, even though the rock is not much more complex than the Health Kit, it *is* somewhat more complex nonetheless. It does less for us in the game, but it is one of those decoration-type items I mentioned—and stuff like this, while unglamorous, can greatly contribute to the ambience of your game.

1. Fire up MilkShape and create an empty document.
2. Use the Sphere tool to create a sphere, as shown in Figure 16.6.
3. In the Side view, select the bottom three rows of vertices.
4. Choose Vertex, Flatten, Y. The bottom three rows should be squished together

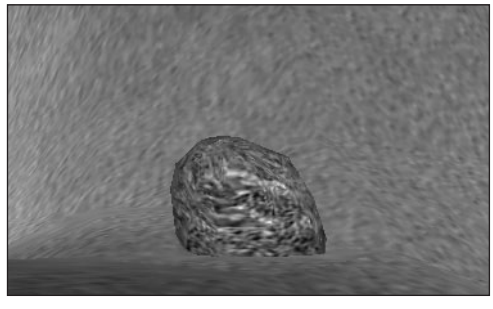


Figure 16.5 The rock in game.

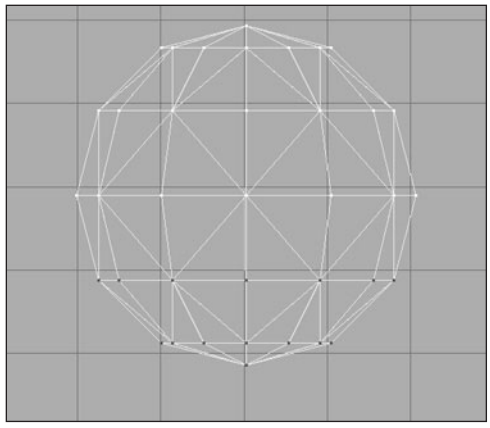


Figure 16.6 The sphere.

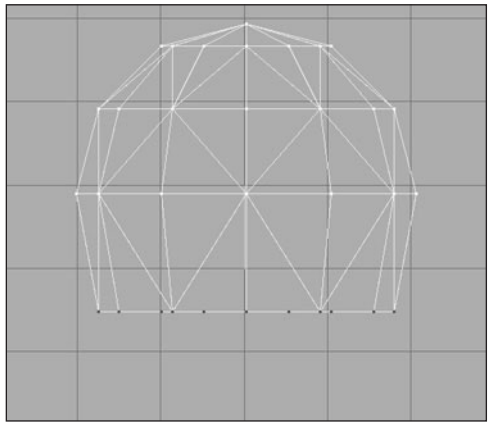


Figure 16.7 The truncated sphere.

in a horizontal flat plane, as shown in Figure 16.7. Already it's starting to look like a rock.

5. Still in the Side view, drag the vertices around on the left side until you get something resembling Figure 16.8.
6. Now in the Top view, drag some more vertices around until you get something resembling Figure 16.9. It's almost a rock now!
7. On the Materials tab, create a new material, using C:\3DGPAi1\RESOURCES\CH16\rock.png as the bitmap.
8. Name the material "rock".
9. Select the entire rock model and assign the rock material to it.
10. Make sure that the 3D view has been set to texture mode. You should see a nice lumpy and ancient-looking rock there, like that in Figure 16.10.
11. Save your work.
12. After saving your work, choose File, Export, Torque Game Engine DTS.

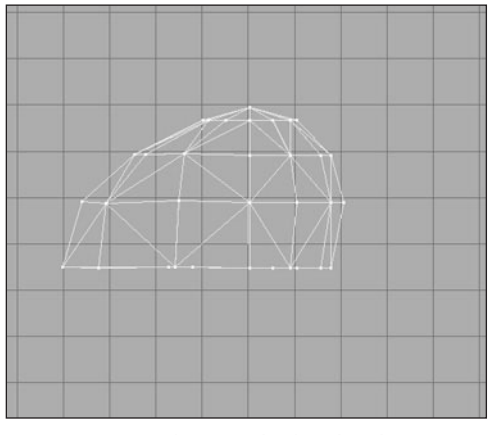


Figure 16.8 The stretched rock-sphere.

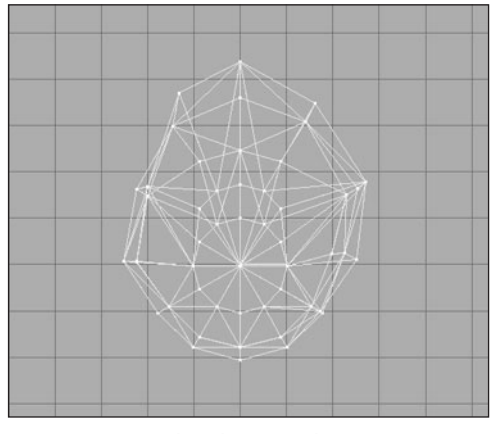


Figure 16.9 The almost rock.

13. You want to take all the defaults (Collision Mesh should be set to Bounding Box).
14. Export the box to C:\3DGPAi1\fps\data\shapes\organic\rock1.dts. Click Yes if you get an alert asking if you want to replace an existing file with that name.

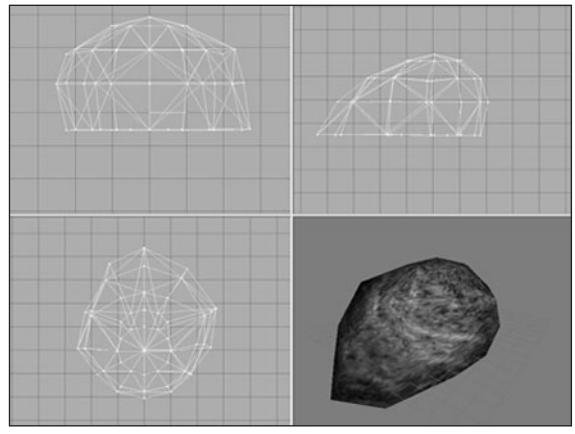


Figure 16.10 The rock model.

Testing the Rock

The rock has a bounding box set for collision because it is, after all, a rock—you can't go through it.

1. Browse to C:\3DGPAi1 and click the Run fps Demo shortcut.
2. Click Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Water World from the mission list.
5. Click Launch.
6. After the game loads, look around for your rock jutting out from a saddle between two hills into the water. Figure 16.11 shows what the rock (marked by the arrow) looks like.
7. Run over to the rock. On top of it you will find the Health Kit.
8. Run over the kit to automatically pick it up. You will see a message in your chat window telling you that you've picked it up.

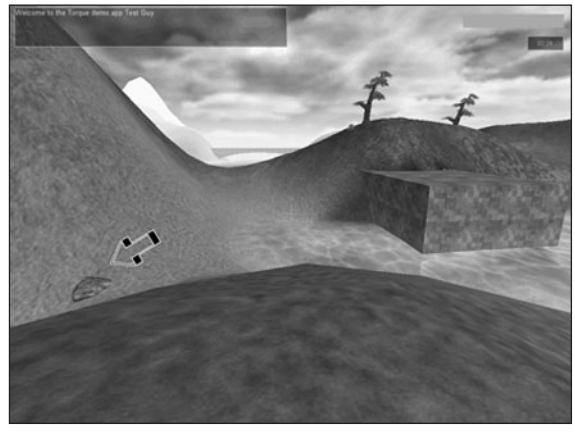


Figure 16.11 Locating the rock.

Trees

If Joyce Kilmer had been a game developer, he might have written, "I think that I shall never see, a model so annoying as a tree." Or something like that. But he didn't, so that's too bad. Really talented game developer poets are rare.

Nonetheless, computer model trees really *are* annoying. There is this

conundrum: If you can interact reasonably well with a model tree, then it looks awful. If the tree looks really good, then interacting with it is awful.

The problem is twofold. We see them everywhere, in most parts of the world, so they provide a great deal of the background to our daily lives. This means that in virtual worlds, if there are no trees in the background, we just *know* something's wrong even if we can't quite put our finger on the problem. They are ubiquitous. And that means we also have a highly developed subconscious sense of what they should look like, when we aren't…um… actually looking at them. With me so far? Okay, that's problem number one.

The other problem is that they are so dad-blamed complex! Even a sapling has lots of little branches and twigs and leaves and buds and stuff. If you have a polygon budget (and if you are making games, *you have a polygon budget!*), then these suckers will dry up that account faster than a barking moonbat can change its mind.

So, on the one hand, to have convincing trees that satiate the subconscious gamer's mind, we need to be attentive to details. And on the other hand, those very details can drag our frame rates lower than a snake's belly in a wheel rut.

To interact well with a tree means several things: When you approach it, circle it, look up into its branches and leaves, you see things properly in three-dimensional perspective. You can collide with the thicker parts like the trunk and the big lower branches, but if you fell onto a tree from above, you would likely fall a long way down through the airy upper structures before you stopped.

But unless you want to put 30,000 polygon trees into your game world, you'll have to compromise on all those fiddly details. There are ways to strip off a few thousand polygons here and there, but long before you get anywhere reasonable, your compromises start making the tree much less treelike. So then you have to pass a few edicts like this:

> *From this point hence, this tree, and all other trees like this tree that grace our fair land, may only be viewed from certain angles—all of the aforementioned angles being from a level on the ground to a level not exceeding the height that one man can jump.*

But then you start to drain the flexibility out of your game world. What if you are standing on the back of a truck? Or on a nearby hill? Or flying overhead in an ornithopter? Well, you can't do any of those things if you really want to save the trees!

And don't even talk to me about having forests of these things in a game. Though there are ways to make trees look absolutely stunning from a distance, and only take two polygons to accomplish the task! Using a technique called *billboarding* we can create trees that look great from any angle as long we are at least moderately far from the tree—say a couple dozen meters or more. But up close they are nothing but flat planes that turn to always face you. You can't look up into them from below and search for robin's nests. You can't climb them. And you certainly can't fall into them from above! I mean, what fun is that?

Team LRN

So why all the blather, you ask? Well, I'll tell you. We're going to look at modeling some game-friendly trees in this section, and I want you to enter into this prepared, understanding why I'm going to show you two different ways to model a tree. There are other ways, but the two represent the opposite extremes. First, we'll create a "normal" low-polygon *solid tree* with a collision mesh. One that you *could* potentially climb using appropriate program code. One that you could actually get beneath and peer up into. It won't look all that great, but it will look like a tree. After that, we'll create a *billboard tree* that can be used to make vast forests of trees that will actually look like forests.

The Solid Tree

The solid tree is constructed of 3D object primitives, mostly cylinders that join end to end and taper. The one we'll make won't have any leaves—it's a generic big backyard tree in the winter.

I should warn you now that we aren't going to build a megapolygon old oak tree or anything like that here. Instead, we're going to do just enough so you'll have a good idea where you can go with the model and what's involved with this approach.

Having said all that, go ahead and create a new empty document in MilkShape, and let's get crackin'.

1. Select the Cylinder tool and set it to 4 stacks and 12 slices in the parameter boxes.
2. Click your cursor in the Side view, and drag down and to the right to create a cylinder like the one shown in Figure 16.12.
3. Still in the Side view, select the vertices in the second row from the bottom, and then use the Move tool to shift them to one side.
4. Switch to the Top view and do the same thing, moving the vertices slightly away from being aligned with the center of the cylinder.
5. Repeat steps 3 and 4 with the next two rows of vertices going up, one row at a time, using Figure 16.13 as a guide.
6. Do an incremental scaling working from the second row of vertices going up, so that you get a tapered trunk, like that shown in Figure 16.14.
7. Use the Duplicate function to make a copy of the trunk.

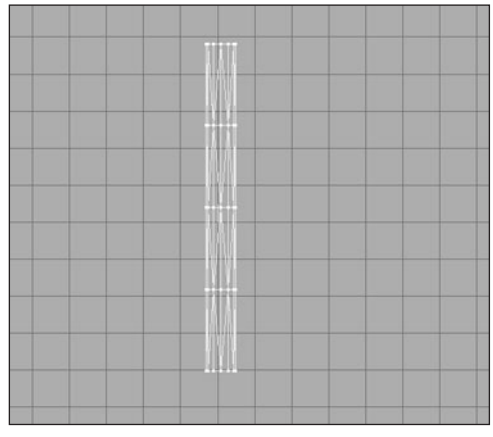


Figure 16.12 Four-stack cylinder.

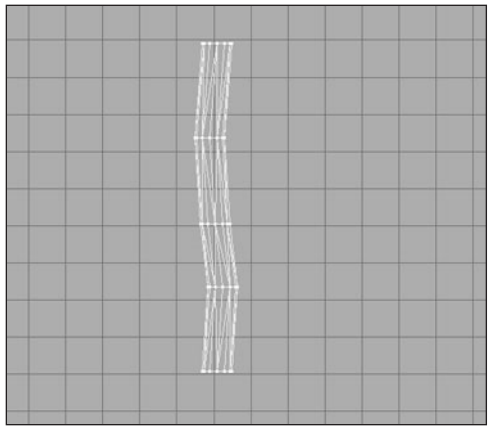


Figure 16.13 Crooked cylinder.

tip

If you've forgotten how to duplicate an object, I'll go over it quickly here.

First make sure that the object to be copied has been selected in face mode. Then choose Edit, Duplicate Selection. Make sure you only do that once, and don't click your mouse in the window.

It will look like nothing happened, but a copy was made in place. Select the Move tool and then drag the selected object to a clear area. There you go—you will have the copy, and the original object will have been left behind.

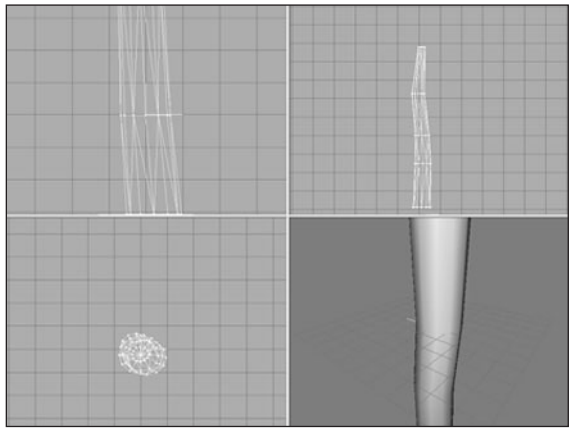


Figure 16.14 Crooked cylinder becomes a tree trunk.

8. Move the copy to one side. Then scale it and rotate it so that you get something that looks like Figure 16.15.
9. Drag the branch over to the trunk and place it with the larger end inside the bounds of the trunk.
10. Make more copies of the branch, scaling, rotating, and tweaking them as desired, until you get something like the model shown in Figure 16.16.

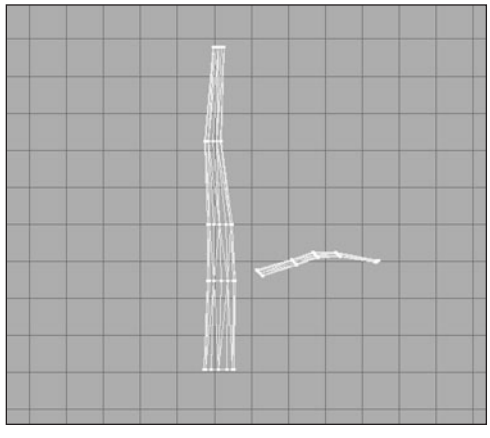


Figure 16.15 Branching out.

11. On the Materials tab, create a new material, using C:\3DGPAi1\RESOURCES\CH16\bark.png as the bitmap.
12. Name the material "bark".
13. Select the trunk using the Groups tab, and assign the bark material to it.
14. Assign the bark material to each of the branches. Do not select them all at once and assign the material—instead, do them one at a time.

15. Make sure that the 3D view has been set to texture mode. You should see the textured tree there, like that in Figure 16.17.

    Okay, we'll stop there. Of course, we could go on and on, making it more detailed, and that's certainly something I encourage you to do. It's just pointlessly repetitive to do it right now. Let's move along and add a collision mesh.

16. Create a box and position it as shown in Figure 16.18.
17. Rename the box, calling it "Collision".
18. Save your work.
19. After saving your work, choose File, Export, Torque Game Engine DTS.
20. You want to take all the defaults (Collision Mesh should set to Bounding Box).
21. Export the box to C:\3DGPAi1\fps\data\shapes\organic\tree1.dts. Click Yes if you get an alert asking if you want to replace an existing file with that name.

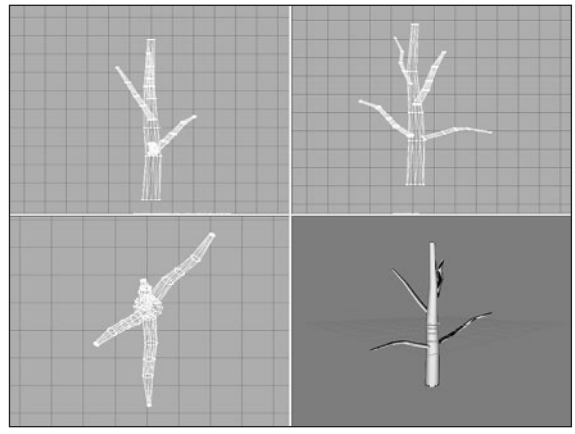


Figure 16.16 Adding more branches.

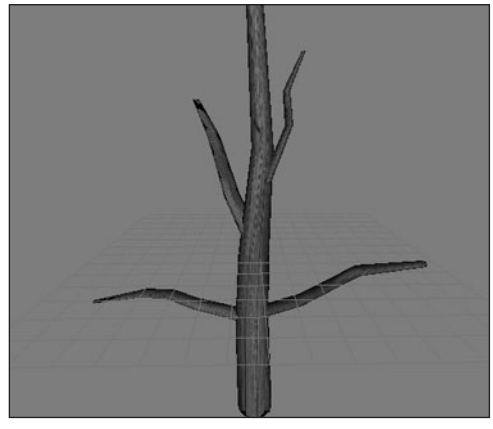


Figure 16.17 The textured tree.

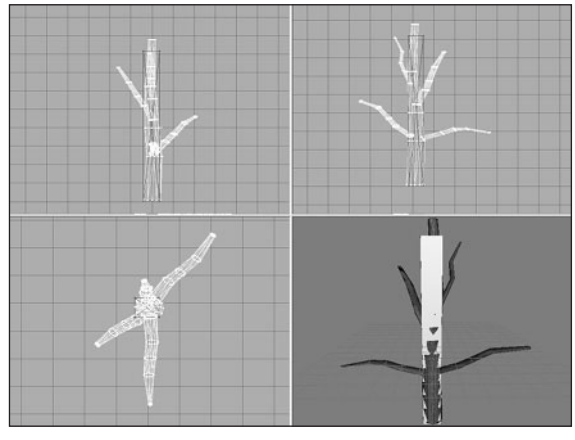


Figure 16.18 The tree collision mesh.

Testing the Solid Tree

The solid tree has a collision mesh—you can't go through it. You also can't climb it as is (you could if you wrote the appropriate script code). Anyway, to test out our solid tree, do the following:

1. Browse to C:\3DGPAi1 and click the Run fps Demo shortcut.
2. Click Start Mission.

3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Water World from the mission list.
5. Click Launch.
6. After the game loads, look around for your tree poking up from a hill behind that block where you found the Health Kit.
7. Run over to the tree and admire your handiwork.
8. Try to run through the tree. If you hurt your head, you know where the first-aid kit is!

The Billboard Tree

The billboard tree is the Ferrari of game trees. It looks good and is so low on polygons that if you remove one polygon, it vanishes! But it is specialized and you can't do much with it.

Create a new empty document in MilkShape.

1. Select the Vertex tool and place three vertices in an equilateral triangle formation.
2. Use the Face tool and create a face, as shown in Figure 16.19.
3. On the Materials tab, create a new material, using C:\3DGPAi1\RESOURCES\CH16\spruce.jpg as the bitmap.

note

There is a quirk in MilkShape when using bitmaps that have alpha channels in them. Because of this, when we make a model that uses a texture with an alpha channel, like the billboard tree, we need to make another version of the texture without the alpha channel, but with the same name (not including extension). So although in the game we will use the spruce.png texture file, we need to use spruce.jpg in the MilkShape model material.

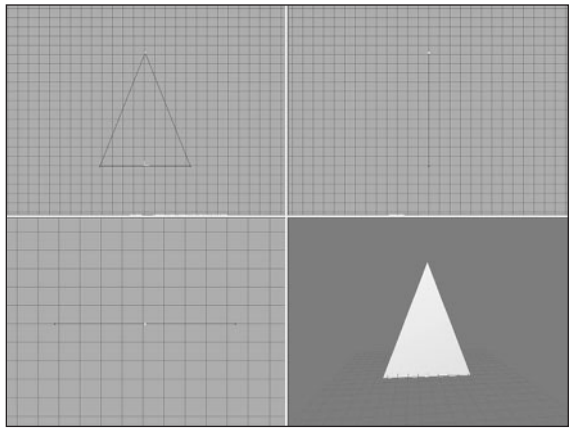


Figure 16.19 The tree triangle.

4. Name the material "spruce: Trans". The ": Trans" part tells the exporter that the alpha channel for this material should be set to translucent mode.
5. With the new material selected, move the right-hand slider to the middle position. This will tell the exporter to enable alpha transparency for this material.

6. Select the triangle using the Groups tab, and assign the spruce material to it.
7. Make sure that the 3D view has been set to Texture mode. You should see the textured tree there, as shown in Figure 16.20.
8. Rename the triangle in the Groups tab as "firstface: BillboardZ". This identifies the polygon as a billboard-z type.
9. Create a *special material*. This is a material that tells the exporter how to do certain things. The directive is embedded in the name—in this case call the material "opt:size=100". There is nothing else needed for the material. This directive tells the exporter to scale the object up by 100 times.

note

A *billboard* is a single polygon object with a texture that always faces toward the player in the game. Some kinds of billboards only face the player horizontally. If you get above them, they don't face up. These are called *billboard-z*.

10. Save your work.
11. After saving your work, choose File, Export, Torque Game Engine DTS.
12. You want to take all the defaults (Collision Mesh should be set to None).
13. Export the box to C:\3DGPAi1\fps\data\shapes\organic\tree2.dts. Click Yes if you get an alert asking if you want to replace an existing file with that name.

Testing the Billboard Tree

The billboard tree has no collision mesh—you can't bump the tree but instead you just go right through it. You also can't climb it, though you probably wouldn't want to.

1. Browse to C:\3DGPAi1 and click the Run fps Demo shortcut.
2. Click Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Water World from the mission list.
5. Click Launch.
6. After the game loads, your tree is actually a group of trees poking up from a hill behind that block where you found the Health Kit.

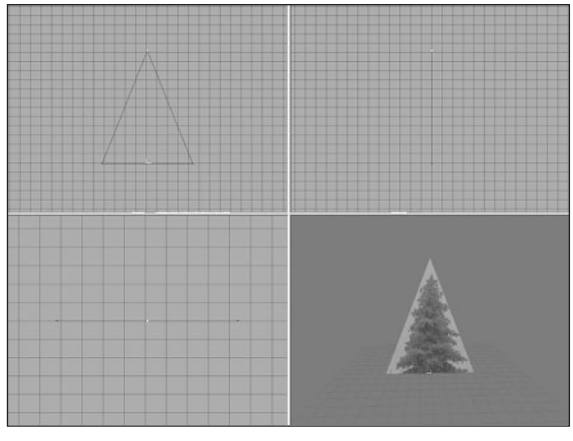


Figure 16.20 The textured tree.

7. Run over to the trees and take a look.
8. Try to run through any of the trees. If you hurt your head, go back and check your model! You may have inadvertently left a collision mesh in it.

As you can see, the billboard tree is kind to your video card and will keep your frame rates right up there. In fact, you can build forests of these things and only have as many polygons as one of the solid trees!

The Tommy Gun

The famous Thompson submachine gun is a somewhat obsolete weapon that most people are familiar with visually, even if they don't know what it's called.

The technique we will use is the Extrusion method. When it comes to modeling weapons, Extrusion is probably the method of choice for the simple reason that it often works well when using photographs for source material. There are dozens and dozens of books and many Internet resources available that have photographs and technical drawings of weapons, but remember that much of the source material is copyrighted.

For our Tommy gun, I'll work from a sketch I made, shown in Figure 16.21.

tip

To create your weapons, you can use a photograph or detailed diagram of your own if you like, however, you are perfectly free to use my sketches and artwork in any way you want to. The choice is yours.

The sketch is rough and not very detailed, but it will do just fine for our purposes. This model will have as few polygons in it as I think we can get away with. I've made two versions of it for you to use: one for the skin and one to act as the extrusion reference image.

Making the Model

Get MilkShape running and warming up in the driveway, and we'll get started in a minute. We will use C:\3DGPAi1\RESOURCES\CH16\tommygun.png as the texture for the Tommy gun's skin. You can find the extrusion guide sketch at C:\3DGPAi1\RESOURCES\CH16\tommygun_ref.bmp. You need to set the latter as the background image for the Side view window in MilkShape; we'll use the former later in this section.

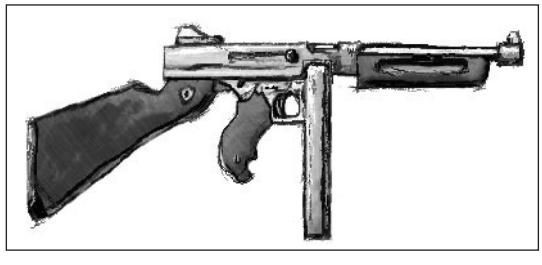


Figure 16.21 Tommy gun sketch.

1. Select the Vertex tool, making sure that the Auto Tool check box is cleared.
2. In the Side view, start placing vertices at all the major corners and points around the components of the gun. See Figure 16.22 for reference.
3. Start making faces. No! I meant in the model, not at me! You will probably have to zoom in some to get enough separation between the vertices.

   Be careful as you move along, making sure you get all the faces. For tips and other information on faces, check back to Chapter 15. Figure 16.23 shows the finished polygon faces around the muzzle, which can be a bit fiddly. Notice how far I zoomed in.

   Figure 16.24 shows the barrel and forestock faces. A warning about the barrel is in order here, I think. In this model we will stick with a straight extrusion exercise—but I highly recommend that after you complete this section you rework the model and make the barrel a cylinder object. The results will be nicer.

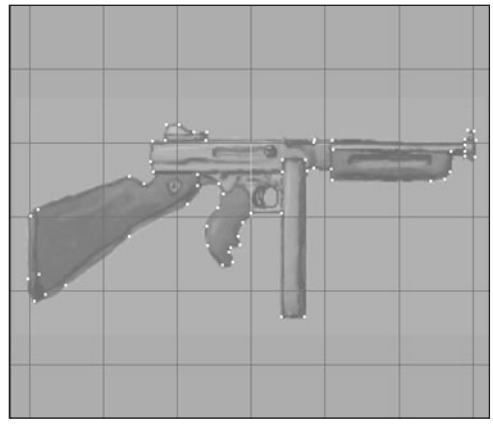


Figure 16.22 Tommy gun vertices.

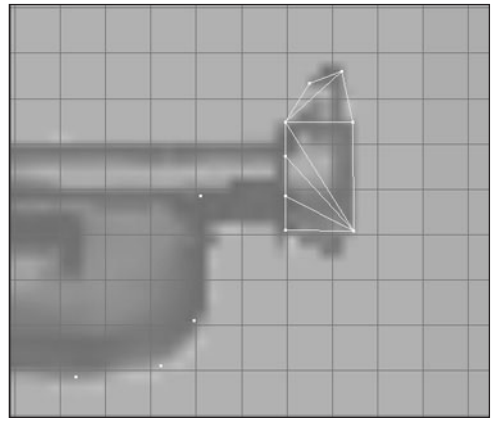


Figure 16.23 Tommy gun muzzle faces.

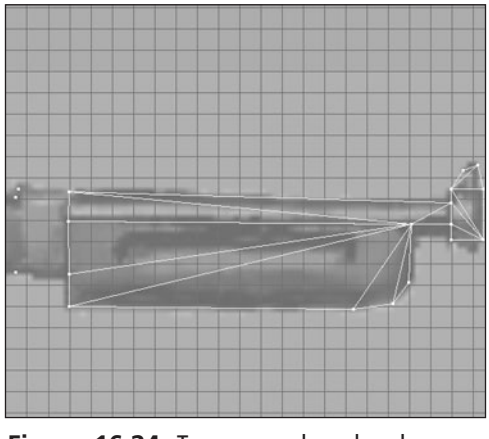


Figure 16.24 Tommy gun barrel and forestock faces.

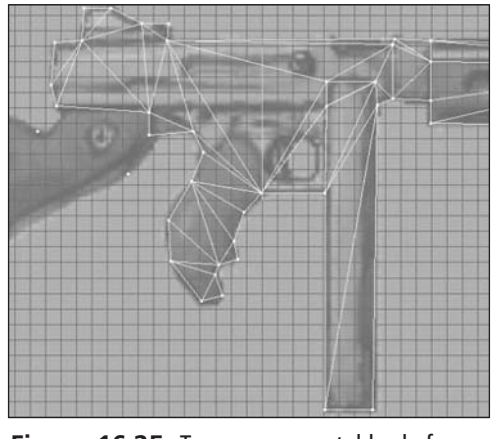


Figure 16.25 Tommy gun metal body faces.

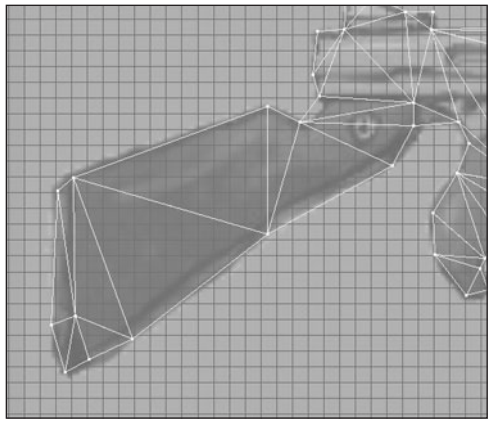


Figure 16.26 Tommy gun shoulder stock faces.

Figure 16.27 Tommy gun loses face—film at 11.

Figure 16.25 shows the faces completed for the grip, the receiver, the magazine, and other metal parts of the main body of the gun. Note that I haven't modeled the trigger or the trigger guard hole—leaving these out saves a ton of polygons. If you want you can add in the detail for the trigger, but you will probably take a hit on frame rate in the game.

Figure 16.26 shows the faces of the wooden shoulder stock.

Now take a look at Figure 16.27. Notice the missing polygon down there in the 3D view? It's not evident by looking at the wire-frame view that the polygon was missed, but its absence really shows in the 3D view. Don't let this happen to you! Heh.

If it does happen to you, fix up the wayward faces and we'll move on to the extrusion.

4. Select all the faces.
5. Use an X value of −2.0 and then click the Extrude button. Do not deselect the highlighted faces after this operation. You should get something that looks like Figure 16.28.

   Next, we will have to cap off one end of the extrusion, like we did with the car model. It's a simple operation but sometimes a bit touchy.
6. Choose Edit, Duplicate Selection.
7. Use the Move tool to drag the copy of the faces back over to the side of the model, using the Front view window to monitor the activity.
8. Zoom in on a few of the vertices in the Front view and make sure that the copy of the faces perfectly aligns with the edge vertices on this side of the model.
9. Select all vertices in the model and then choose Vertex, Snap To Grid. One or two of the vertices might snap to an awkward location, so go ahead and manually fix them.

10. Choose Vertex, Weld Together.

    The model as built so far is fine, except that it was created at a scale four times larger than we want for use in game. This was deliberate—a larger scale allows us to use larger reference images for the background image, which gives us access to more detail. Also, the larger the scale, the finer the granularity available when we want to snap points to the grid. So after all this work, we need to scale the model back to the correct size.

11. Select all parts of the model.
12. Use the Scale tool to set the scale to 0.25 in all three axes.
13. Click the Scale button to the right of the axis boxes. The gun will shrink and should appear roughly as shown in Figure 16.29.

    Next, we have three nodes to add—one to indicate where the gun is held, one to indicate where the muzzle is, and one to indicate where expended shells are ejected. These nodes inform the engine where these spots are; a script that will be defined later dictates how they are used.

14. Create three unconnected joints, positioned and named as shown in Figure 16.30. The three node names are mountPoint, ejectPoint, and muzzlePoint.

    We have one more thing to do. We need the gun to have the correct posture when held by the player model.

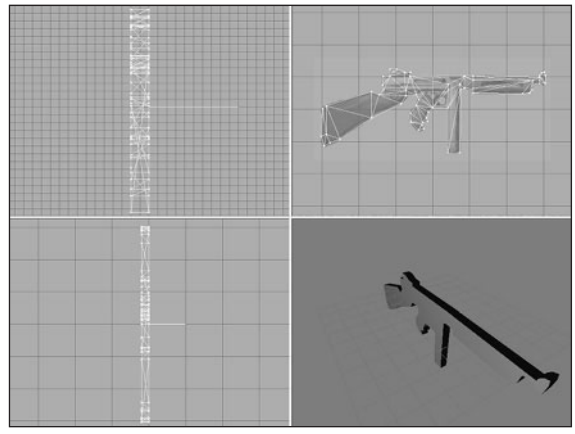


Figure 16.28 Extruded Tommy gun.

Figure 16.29 Shrunken gun.

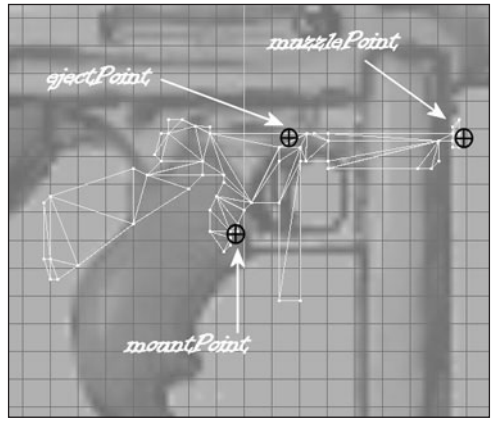


Figure 16.30 The Tommy gun nodes.

15. In the Top view, select all faces.
16. Rotate the gun about 8 degrees to the left, as shown in Figure 16.31.
17. Move the nodes to align them with the gun, using Figure 16.31 as a guide.

Voilá! Insta-gun. Save your work.

Skinning the Tommy Gun

Way back in Chapter 13 you learned how to use UVMapper and Paint Shop Pro to create a skin for objects. In this chapter we'll look at using the built-in Texture Coordinate Editor in MilkShape to accomplish the same thing. It can be awkward to use but is suitable for our purpose here because we will already have a texture to use for the skin—in this case we will use a version of my original sketch.

1. Create a new material, using the file tommygun.bmp as the bitmap for the texture.

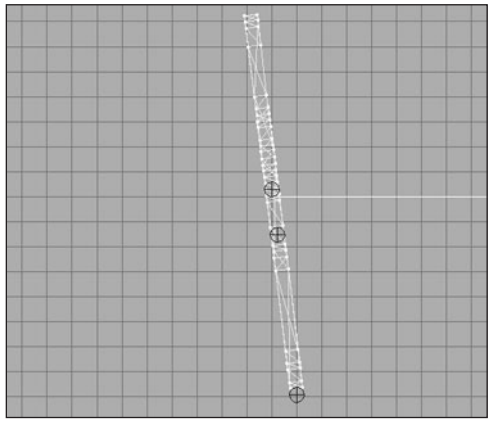


Figure 16.31 The rotated Tommy gun.

2. Assign the new material to the Tommy gun object.

   If you have more than one object, select all of the faces in all of the objects, then regroup them. After that, you can assign the new material to the single object.
3. Use the Groups tab, and select the Tommy gun object.
4. Choose Window, Texture Coordinate Editor. You will get the Texture Coordinate Editor dialog box, as shown in Figure 16.32.

Figure 16.32 Texture Coordinate Editor dialog box.

   When the Texture Coordinate Editor first opens, you often see just some confusing white lines over the top of the texture assigned to the object you are skinning. Not to worry.
5. Select the appropriate view from the view selection combo box at the right side of the dialog box. In the case of the Tommy gun, this is the Left view.
6. Click the Remap button. You should get something like that shown in Figure 16.33. The shape

of the Tommy gun may not line up with the texture, so go ahead and use the Select and Move buttons inside this editor to move the vertices until they are in place. Figure 16.33 is a good guide to what the final result should look like.

7. Close the Texture Coordinate Editor dialog box and take a look in your 3D view (make sure that it is set to texture mode). There it is—your Tommy gun! Compare your work with Figure 16.34. As you examine it closely, you will see why it might be a good idea to redo the barrel as a cylinder.

Testing the Tommy Gun

In order to test the Tommy gun, we first need to export it from MilkShape.

1. After saving your work, choose File, Export, Torque Game Engine DTS. You will see the Torque Game Engine (DTS) Exporter appear.
2. Use the defaults, but make sure they are correct.

   You want to have Export Animation and Export Material Information enabled, and the Collision Mesh set to None (this is automatic if the exporter finds that there is not a mesh whose name starts with "Collision"). Items that are used as weapons don't need to have a collision mesh created—the Torque Engine knows how to handle them without the collision mesh. Click OK when ready.
3. Export your Tommy gun to DTS format as C:\3DGPAi1\fps\data\shapes\Tommygun\Tommygun.dts.

Figure 16.33 Remapped view.

Figure 16.34 Finished Tommy gun.

Next, you need to edit the script that controls the weapon so that it will look for your model, and not the default one. We'll use the simple futuristic blaster-style rifle in the fps demo that comes with Torque. It might sound funny, and the bullets will look weird, but that's okay—we just want to see our Tommy gun model in a gamelike environment.

1. Locate the file C:\3DGPAi1\fps\server\scripts\rifle.cs and open it with UltraEdit.
2. Find the line that says this:

   ```
   shapeFile = "~/data/shapes/buggy/buggy.dts";
   ```

   and replace it with this line:

   ```
   shapeFile = "~/data/shapes/car/car.dts";
   ```

3. Then find the line that says this:

   ```
   pickUpName = "a rifle";
   ```

   And replace it with this line:

   ```
   pickUpName = "a tommygun";
   ```

4. Save the file.

Okay, now it's time to run the fps demo.

1. Browse to C:\3DGPAi1 and click on the Run fps Demo shortcut.
2. Click Start Mission.
3. In the Launch dialog box, make sure that the Multiplayer Mission box is cleared.
4. Select Water World from the mission list.
5. Click Launch.
6. After the game loads, look around for a big gray block jutting out from the side of a hill into the water. Figure 16.4 in the earlier section about the Health Kit shows what the block (marked by the arrow) looks like.
7. Run over to the block. On top of it you will find the Tommy gun.
8. Run over the gun to automatically pick it up. You will see a message in your chat window telling you that you've picked it up.
9. Look elsewhere on the block—there are ammo boxes at the other corner. Go pick them all up by running over on top of them.
10. Find something to shoot at and let go!

After a short while, a new copy of the Tommy gun will reappear at the spot where you picked up the first one. Head back over there so you can admire your modeling handiwork as it gently rotates in the, errr…breeze. Yeah, a circular breeze—that's what it is!

There is also a primitive crossbow located on the bridge, if you are of the mind to go and check it out. Press the 1 key or the 2 key to switch back and forth between the weapons if you have more than one.

The Tommy Gun Script

Just as we encountered with the runabout in Chapter 15, making the model for a weapon is not the whole job. We have yet to create the weapon script that defines how the weapon works. That is something we will cover later, in Chapter 22, when we look at the code that brings all of these models together in our sample game.

Moving Right Along

This chapter was a bit of a review of techniques covered in earlier chapters, but we applied these techniques to a few different kinds of items with different features: collision meshes, no collision meshes, billboard textures, and translucent textures. This helps demonstrate the wide variety of characteristics that modeled items can have in a game world.

We also looked at the MilkShape's Texture Coordinate Editor a little bit, enough to be able to use it to tweak texture maps if the situation warrants.

In the next chapter, we'll learn a new sort of 3D modeling tool and use it to make structures.

This page intentionally left blank

Team LRN

CHAPTER 17

Making Structures

There are quite a few different options available to the game developer when creating objects for use in a game. We've already seen how to make things like trees and weapons using MilkShape to create DTS-type objects. But what about complex structures, like buildings and bridges?

Well, you can also use DTS objects for those, making sure to create multiple collision meshes where needed. Buildings have a lot of surfaces that either you can walk on or at the very least you can't walk *through*. So you will spend a great deal of effort creating collision meshes. Also, DTS objects don't inherently understand the concept of lighting and shadows, so if you make a building as a DTS object, you will also have to light it yourself using another object—which is possible, but you'll have to do that for every building. This is a real issue for structures that are interiors. Bridges and ramps are not such a big deal when it comes to lighting, but you still have the collision mesh workload.

Fortunately, there is a solution to this problem. In Torque a different kind of object is supported—a DIF-type object, also called an *interior*. Now, using the word *interior* is a bit misleading, because you could, and probably will, use the same kind of object for complex structures that don't have interior lighting but do have many collidable surfaces. Therefore, I prefer to use the word *structure* to describe DIF objects.

There are several tools available to use for creating DIF objects. A very good open source (it doesn't cost you anything to use and it is published under the GNU General Public License) is QuArK (short for Quake Army Knife).

Team LRN

Installing QuArK

All of the parts you will need to install QuArK on your hard drive are included on the companion CD. After installing the program, you'll need to verify the configuration, as described in the following sections.

Using the Installer

When you install QuArK, you will also be installing a stripped-down version of the Python scripting language.

Do the following:

1. Browse to your CD in the \QuArK directory.
2. Locate the Setup.exe file and double-click it to run it.
3. Click the Next button for the Welcome screen.
4. Follow the various screens, and take the default options for each one, unless you know you have a specific reason to do otherwise.

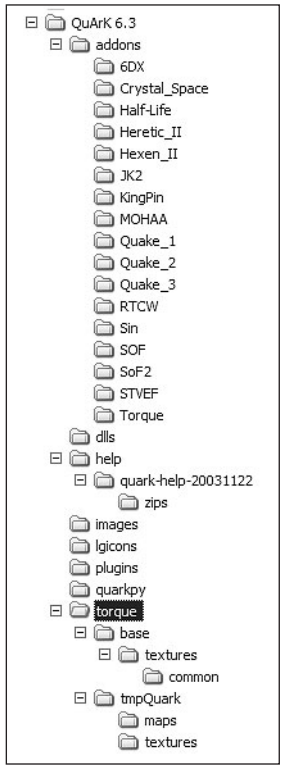


Figure 17.1 QuArK directory structure.

Note that this is not the official QuArK installer—there isn't any. This installer is only available on the companion CD for this book.

After installation you will have QuArK installed into a directory structure that should look exactly like that in Figure 17.1.

Configuration

The installation program will have already ensured that you have the correct configuration for use with this book. However, it's a good idea to double-check the configuration.

1. Launch QuArK by choosing Start, Programs, 3DGPAi1, QuArK, Launch QuArK.
2. After the QuArK window appears, choose Edit, Configuration (see Figure 17.2).

   The Configuration utility will appear. On the right side is a large blank area where various parameters will appear to be edited, depending on what you choose in the left area, which contains the configuration categories.
3. Click the Torque category, located in the Games section (see Figure 17.3).

   The parameter page for Torque will appear, as shown in Figure 17.4.

4. Verify the setting of each parameter by checking it against Table 17.1. All parameters not listed in the table should be left blank. Also note that the settings in the first program through the sixth program entries are almost identical, with the only difference being the middle of the three command-line switches in the fixed command-line arguments parameter for each program.
5. Close the Configuration utility when you are finished.

Next, let's make something!

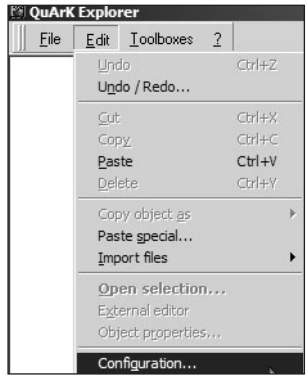


Figure 17.2 QuArK Edit menu.

Quick Start

What we'll do is fire up the QuArK Map Editor and quickly create a structure that we can stick in our sample Torque game and poke around with.

note

More taxing terminology! In QuArK development circles what you create are often called *rooms*. This harkens back to QuArK's beginnings as an editor for Quake, where everything was a room. There were no outdoor areas as such—no external terrain. When we create the rooms and save them, we save them as map files, because Valve used the word *map* to describe their version of what id Software (the guys who made Quake) called a *room*. Clear as mud?

And of course, just to be difficult, GarageGames calls these creations interiors (which fetches back to the term rooms in a way) in Torque. I use the word structure, which I think is both pithy and generic at the same time. My use of structure can encompass room, map, and interior as they are used in their respective contexts while still also applying to things like bridges and guard towers.

So room = map = interior = structure. In any event, the source format for the files we will be dealing with are MAP, a text file format, while the compiled version is called DIF, a binary format. We save our work as .map and compile the work to .dif for use in Torque.

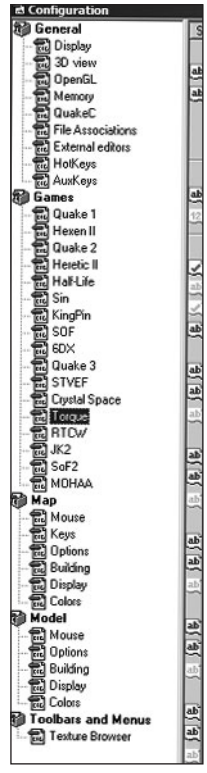


Figure 17.3 Configuration sections and categories.

1. Start the QuArK Map Editor by first selecting what game you are working with in the Games menu. For now click new Torque map, as shown in Figure 17.5.

   You will then get the Map Editor, as shown in Figure 17.6. A default structure (or *room*, remember?) will appear, all ready for you to hack away at. This is convenient, because you will probably be creating roomlike structures most of the time anyway. This makes a useful starting place.

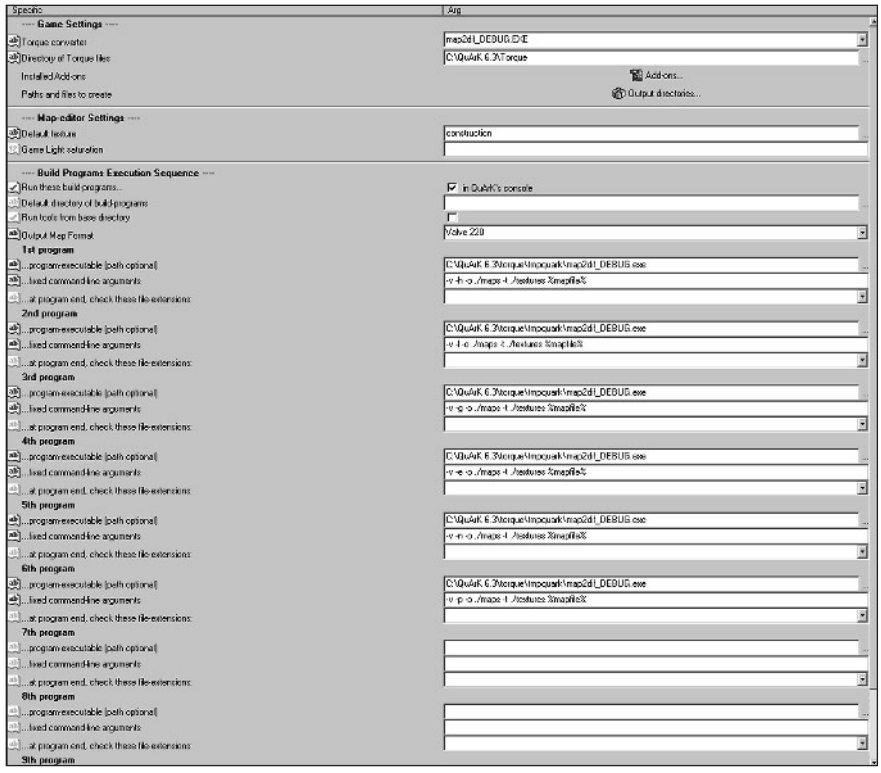


Figure 17.4 Torque configuration page.

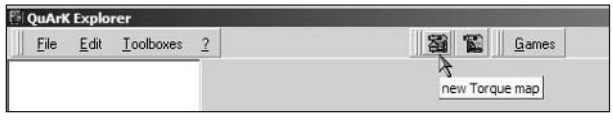


Figure 17.5 The Torque button in QuArK.

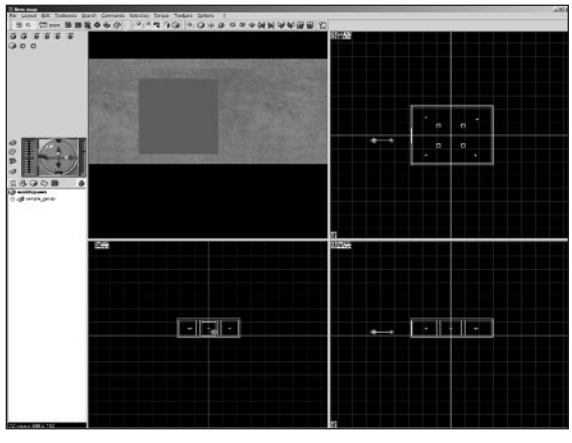


Figure 17.6 The Torque Map Editor in QuArK.

tip

To delete the default room, locate the Tree view on the left side of the Map Editor window, and select the sample_group entry, as shown in Figure 17.7.

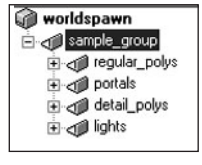


Figure 17.7 The Tree view.

You can press the Delete key or right-click on the entry to get the pop-up menu. Then choose Edit, Delete.

You should now have a blank slate.

Table 17.1 Torque Settings for QuArK

Parameter	Value
Torque converter	map2dif_DEBUG.EXE
Directory of Torque files	C:\QuArK 6.3\Torque
Run these build programs	selected (in QuArK's console)
Run tools from base directory	cleared
Output Map Format	Valve 220
1st program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -h -o ./maps -t ./textures %mapfile%
2nd program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -l -o ./maps -t ./textures %mapfile%
3rd program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -p -o ./maps -t ./textures %mapfile%
4th program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -e -o ./maps -t ./textures %mapfile%
5th program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -n -o ./maps -t ./textures %mapfile%
6th program	
program executable (path optional)	C:\QuArK 6.3\torque\tmpquark\map2dif_DEBUG.exe
fixed command-line arguments	-v -g -o ./maps -t ./textures %mapfile%

2. Choose from the menu Torque, Export mapfile only. This creates newmap.map.
3. Now, from the menu, choose Torque, Build DIF only, as shown in Figure 17.8.

   You will see a console window appear, and a bunch of stuff will be written into it. This is the output of the map2dif_DEBUG.exe map compiler program that comes with Torque. We are using the debug version because its output provides a lot more information that could help solve problems that we can encounter with more complex maps. The program takes newmap.map and compiles it into DIF form.

   Anyway, you should see a line near the bottom of the output that reads something like this:

```
Writing Resource: persist..(./maps/newmap.dif) Done.
```

Team LRN

If you see this, your map has successfully compiled.

4. Next, from the same menu, choose Torque, Prepare used textures.

   This will create a list of all the textures used in your creation and copy them to the prepared textures directory. After a brief pause, you will get an information alert, like that shown in Figure 17.9.
5. Browse to your map output directory, C:\QuArK 6.3\torque\tmpquark\maps, and locate newmap.dif.
6. Copy newmap.dif to your Emaga6 sample game, into the directory C:\aEmagaCh6\control\data\structures.
7. Browse to your prepared textures directory, C:\QuArK 6.3\torque\tmpquark\textures, and locate concrete.jpg and NULL.jpg.
8. Copy concrete.jpg and NULL.jpg to the directory C:\aEmagaCh6\control\data\structures.
9. Use UltraEdit to open the mission file for Emaga6, C:\aEmagaCh6\control\data\maps\book_ch6.mis.
10. Locate the first instance of this line:

```
interiorFile = "~/data/structures/hovelb.dif";
```

    and change it to the following, to reflect your new structure:

```
interiorFile = "~/data/structures/newmap.dif";
```

11. Directly above that, find this line (the numbers might be different):

```
position = "-4.05031 54.9271 207.919";
```

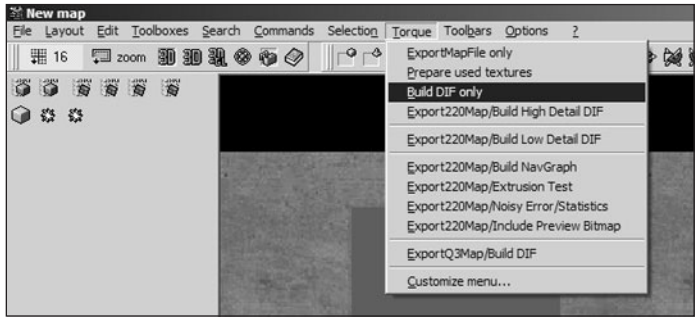


Figure 17.8 Selecting Build DIF only.

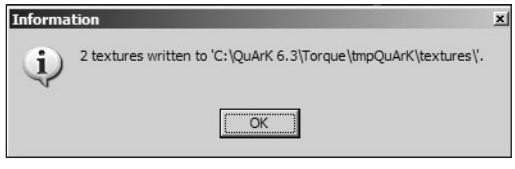


Figure 17.9 The texture information alert.

and replace it with this exactly:

```
position = "8 50 198";
```

The preceding steps are a way to manually insert your creation into the mission map file at the location of one of the existing hovel structures. Because of differences in construction between the two structures, you needed to adjust the position of the new structure somewhat.

12. Run your Emaga6 sample game. After spawning in, you should see the new structure directly in front of you, as shown in Figure 17.10. Run on over and go inside. Take a good look around. There's your first structure created with QuArK!

Building Bridges

So, you've had a taste of how you can use QuArK. Now let's dive in and muck about with it a bit and actually create something. Because this is our first structure from scratch, we'll start out with something not too complex—a stone bridge.

1. Launch QuArK and open the Torque Map Editor, as you learned in the last section.
2. Delete the default room.

note

QuArK and other similar programs that are based on constructive solid geometry use something called *brushes* to create their objects. A brush in this sense is like a building block. You select a particular brush (also called a *polyhedron* in QuArK) for a particular need and apply it to your model.

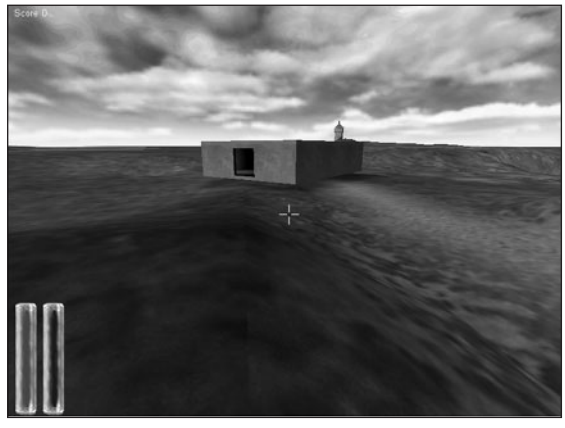


Figure 17.10 The test structure in Emaga6.

3. Select the Cube brush by clicking its icon, as shown in Figure 17.11.

   After clicking the Cube brush icon, you will see a cube appear in your model, as shown by Figure 17.12. Notice that it has a small solid square in the center, called the *handle*, and small "empty" squares on each edge, called *face handles*. You use the face handles for resizing the brush and the center handle for moving the brush. Simply click the mouse cursor

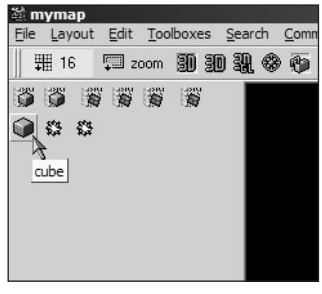


Figure 17.11 The Cube brush.

in the handle, hold down the button, and then drag the brush around to resize or move it as needed.

Also, you should note the texture that the brush sports—a striped texture. This is the default texture for new brushes; the texture's name is Construction. You will assign your own textures later.

4. Using the handles, resize and move the brush until you have something resembling Figure 17.13. This is the roadbed of the bridge.
5. Create two more Cube brushes and place them as shown in Figure 17.14. These will be the bridge pylons.

   Next, you are going to add some texture to the bridge.
6. Click inside the Roadbed brush to select it.

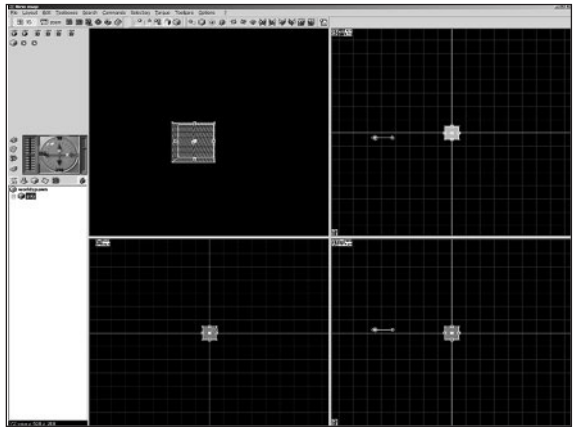


Figure 17.12 A new cube.

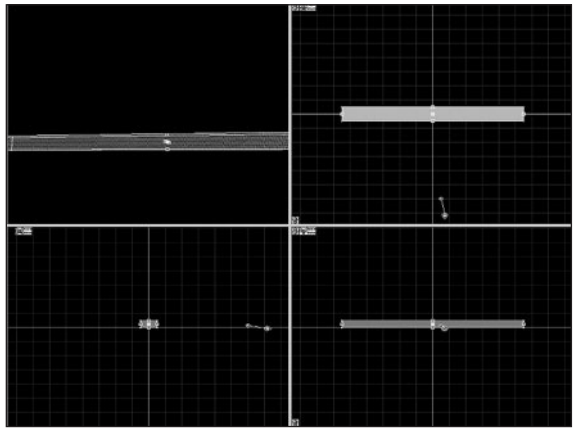


Figure 17.13 Reshaped cube.

7. Right-click anywhere in the view to get the pop-up menu. As long as you have a brush selected, you will get a pop-up menu containing functions that will be applied to that brush, even though you haven't actually right-clicked on the selected brush.
8. Choose Texture, as shown in Figure 17.15.

   The Texture Browser will appear, as shown in Figure 17.16. Locate the Book folder and select the 256_BrickA texture. You can see any of the textures in the preview pane at the right by selecting the texture in the browser frame on the left.
9. Double-click the 256_BrickA texture, and the Texture Browser window will go away. Your texture will be applied to the selected brush.
10. Add the 256_BrickA texture to all of the other brushes in the bridge model.

You will end up with a textured bridge (see Figure 17.17) in your 3D view at the upper left.

11. For safety's sake, save your work, by Choosing File, Save As File. In the Save As Type combo box, select Quake map file (*.map).
12. Choose Torque, Export220map/Build High Detail DIF.

    This will export the map again, overwriting what you just saved (so you can see that you could have gotten away with not doing the initial Save As operation), and then compile the map to DIF format and gather all of the used textures into the used textures directory. Contrast this all-in-one operation with the discrete steps we executed earlier when we did the quick start with the default room. This is certainly more convenient!
13. Copy the textures and the bridge.map file to C:\aEmagaCh6\control\data\structures.

    There is already a bridge with that name in that directory that is being used in the mission file. Go ahead and replace that file, as well as the textures, if necessary.
14. Launch Emaga6 and when you spawn in, turn to your right and run to the edge of the wadi. You should see your bridge spanning the wadi ahead of you, as depicted in Figure 17.18.

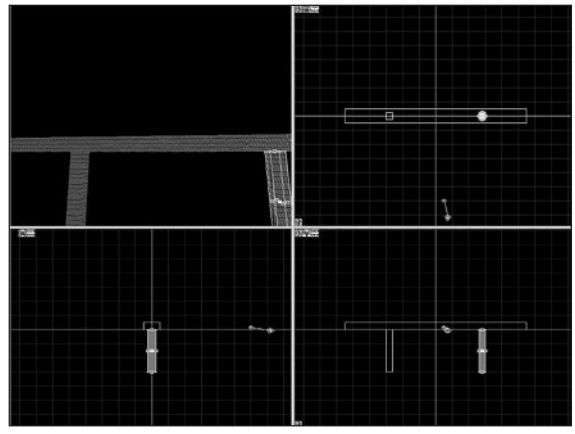


Figure 17.14 A new cube.

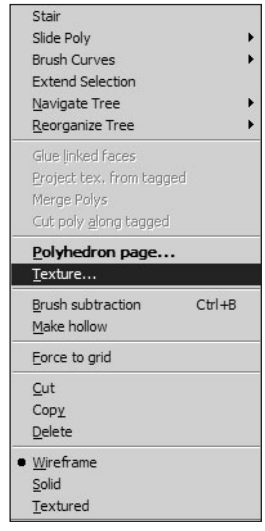


Figure 17.15 The Texture item in the Object pop-up menu.

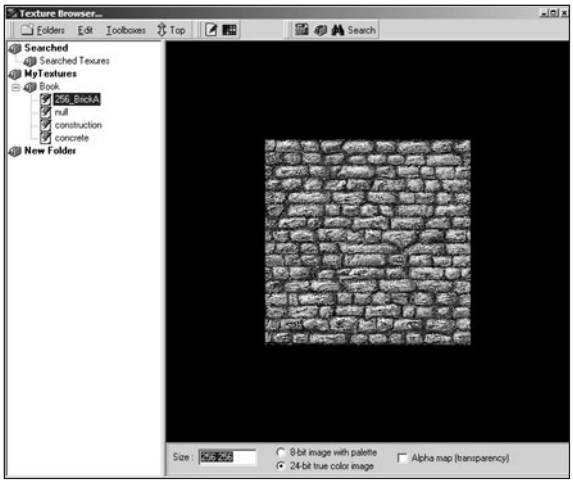


Figure 17.16 The Texture Browser.

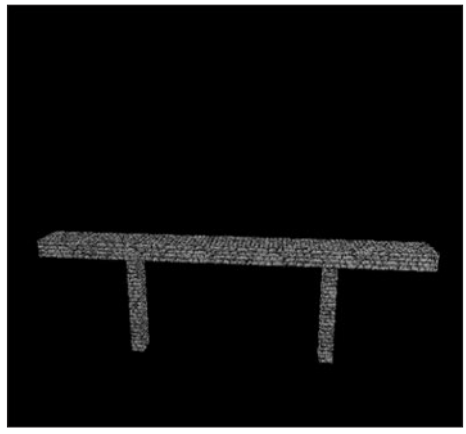


Figure 17.17 The finished bridge.

Building a House

The bridge was nice and certainly useful. But one must admit that it's fairly simple. Of course, if you need a bridge you will probably make something more ornate.

In this section we will go a little bit farther and make something more complex: a house with a door opening and a window and internal lighting.

1. Launch QuArK and open the Torque Map Editor. Delete the default room and save the empty file as C:\QuArK 6.3\torque\tmpquark\maps\house.map.
2. Select the Cube brush and create a cube that measures 260 units in width by 360 units in length by 256 units in height. You can see the dimensions by hovering the cursor over the cube and looking in the lower-left corner of the Map Editor window.

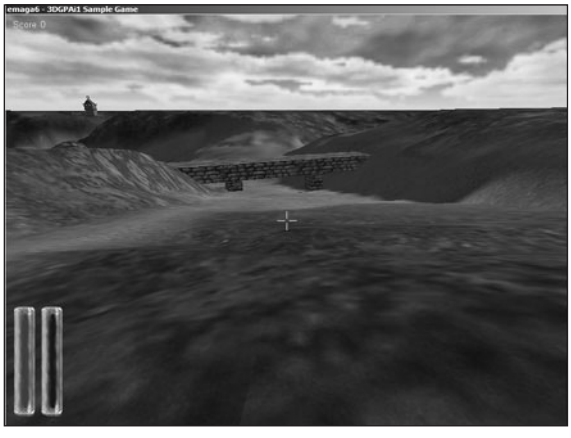


Figure 17.18 The bridge in Emaga6.

3. Create a smaller brush in front of the first cube, as shown in Figure 17.19. This will be used to create the front stairs.
4. Select the new brush in the lower-left window, and then right-click to get the pop-up menu and choose Stair. You will see your brush change into a staircase, as shown in Figure 17.20.
5. Select the large brush, right-click to get the pop-up menu, and choose Make Hollow. Your large

brush will now be a hollowed room and will look like the one in Figure 17.21.

6. Create another brush and place it in the wall above the stairs, as shown in Figure 17.22. This brush will be used for an operation called a *subtraction*.
7. Ensure that the new brush is selected, right-click it, and choose Brush Subtraction. You will see a bunch of new lines appear in the wall in which the brush is embedded.
8. Select the Subtraction brush and press the Delete key to make it go away. You will see a hole in the wall now, as depicted in Figure 17.23. This is the doorway.
9. Using the texture assignment technique you learned in the earlier sections of this chapter, set the house's texture to concrete and the stairs' texture to 256_BrickA.

   Now you have to add a special brush of a type known as an *entity*. Entities are constructs used when making maps that give special instructions to the map compiler about things it needs to do or understand when it goes about building the DIF version of the map for use in Torque.

   You are going to create an entity called a *portal*. Portals tell Torque how to go about lighting the interior of the object when

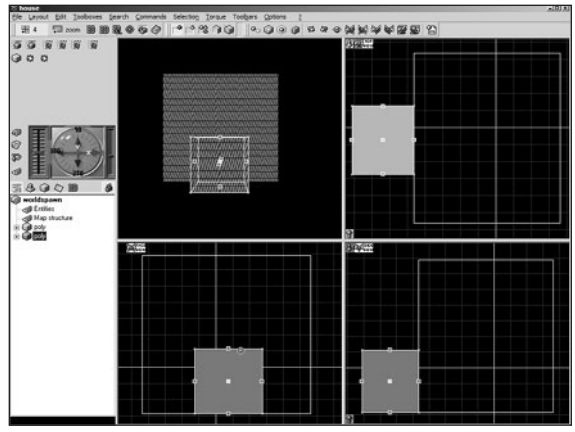


Figure 17.19 The Stair brush.

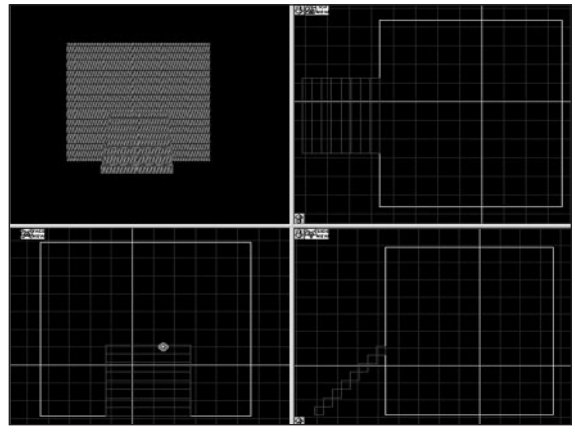


Figure 17.20 The stairs.

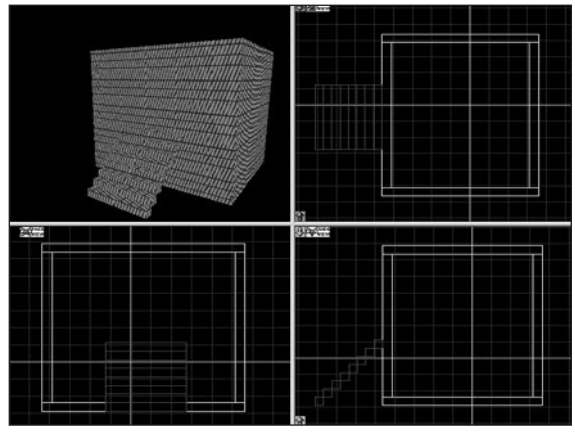


Figure 17.21 The hollowed room.

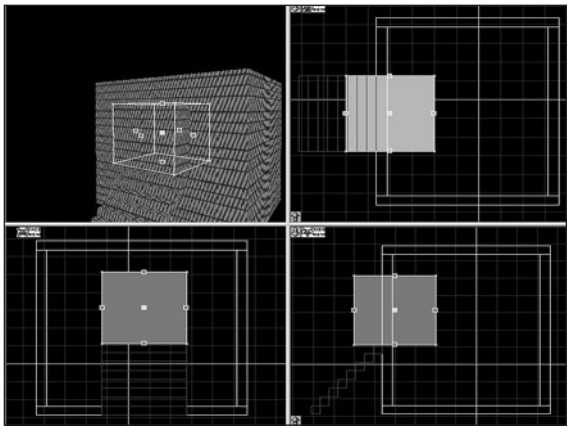


Figure 17.22 The Subtraction brush.

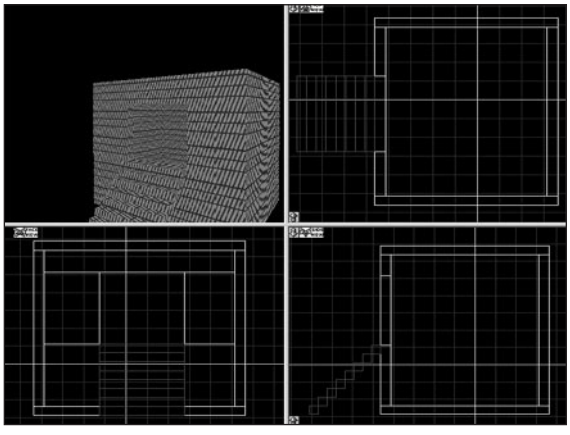


Figure 17.23 The doorway.

there is an opening. By placing the portal in the doorway, you can tell Torque that it should not try to do its special interior lighting outside the door. You can also indicate whether or not Torque will allow external light to pass through the portal to the interior.

10. Locate the Entities panel in the upper-left portion of the Map Editor, and in it find the Torque Entities button. Click it.
11. Choose Brush-based entities, portal, as shown in Figure 17.24.
12. An entry for the entity will appear in the Tree view in the lower-left portion of the Map Editor, as shown in Figure 17.25.
13. Select the portal entity and then click the Cube button to create a Portal brush, as shown in Figure 17.26. A new brush will appear in the model—this is the Portal brush.
14. Reshape the Portal brush to be just a little larger than the doorway in width and height, and a little narrower than the doorway in thickness, as you can see in Figure 17.27.
15. In the Tree view, double-click the portal icon (not the poly icon that it contains). A small dialog box will appear; the bottom field is the parameter `ambient_light` setting for the portal. Choose passes through from the combo box.
16. Using the texture assignment technique you learned in the earlier sections of this chapter, set the portal's texture to `NULL`.

    Next we are going to create another entity—a light entity. The process is similar to the portal entity at first, but a light entity is a *point* entity, not a brush entity, so things will be just a little different.

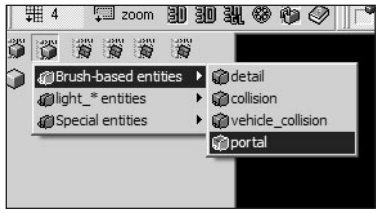


Figure 17.24 The portal entity.

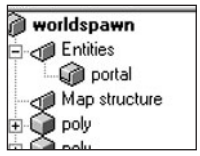


Figure 17.25 The portal entity in the Tree view.

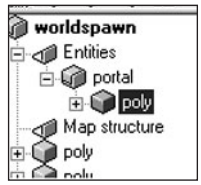


Figure 17.26 The portal entity's brush in the Tree view.

17. As before, go to the Entities panel and find the Torque Entities button. Click it and choose light_*entities, light_omni, as shown in Figure 17.28.
18. A new light entity will be created in the Tree view, and a small rendition of a light bulb will appear in your map. Move it up closer to the ceiling, as you can see in Figure 17.29.
19. Double-click the light_omni entity in the Tree view to get to its settings. Ensure that alarm_type is set to normal only.
20. Save your work and export it using the methods you learned earlier in the Quick Start section.
21. Use UltraEdit to open the mission file for Emaga6, C:\aEmagaCh6\control\data\maps\book_ch6.mis.
22. Locate this line:

```
interiorFile = "~/data/structures/newmap.dif";
```

If the line doesn't exist in the file, then locate this one:

```
interiorFile = "~/data/structures/hovelb.dif";
```

Change whichever line you find to reflect your new structure as follows:

```
interiorFile = "~/data/structures/house.dif";
```

23. Run Emaga6 and check out your new house, which should be in front of you. Go on inside and see the effect the lighting has.

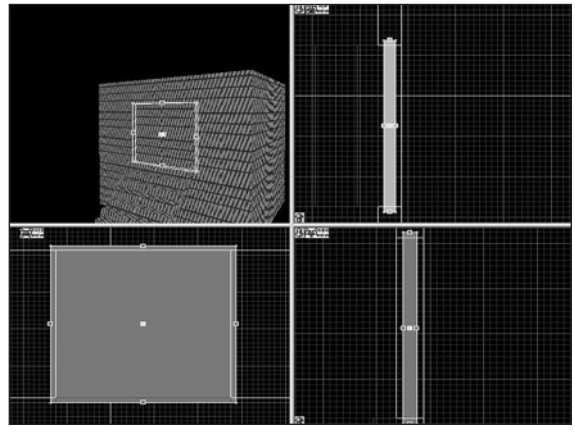


Figure 17.27 The portal in place.

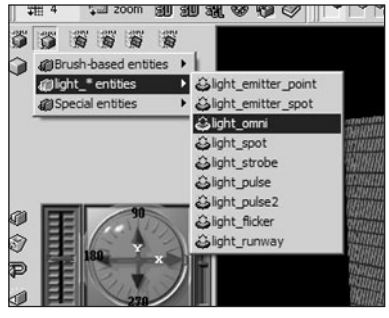


Figure 17.28 The light entity.

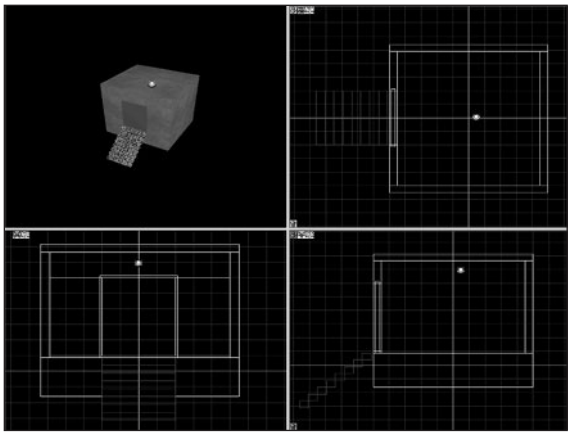


Figure 17.29 The repositioned light entity.

Moving Right Along

So, in this chapter you've learned yet another tool. QuArK is an extremely feature-complete tool for creating structures for Torque. You've built the two most common sorts of structures, an outdoor structure (the bridge) and a building with a lighted interior.

Your imagination is the only real limit here—castles, complex underground tunnel systems, factories, playgrounds, and just about anything else can be created with QuArK.

Normally, I would include a reference section for QuArK in this chapter. However, the program has so many features and options that the material is just too hefty to present in the chapter. Instead, I've included the QuArK reference in Appendix D.

In the next chapter, we'll take a look at how to make things for the game world environment.

CHAPTER 18

Making the Game World Environment

In many games, having a full suite of character models, buildings, trees, and other visual clutter is still not enough to accomplish the needed sense of immersion. There are a number of other aspects to the game world that come from the world around us that we often take for granted: the background sky, the appearance of water, the appearance of clouds in motion, and the terrain. Figure 18.1 is a nice serene picture of ocean-side forested hills just after sunset. No, it's not a photograph—it's a screen shot from the game *Tubettiworld* being developed using the Torque Game Engine.

Now way back in Chapter 12 we covered terrains to a certain extent, so you probably have a reasonable sense of what is involved with creating terrains using a height map. In this chapter we will revisit terrains using the more labor-intensive method of manually building up a terrain with the in-game editor. We'll get into that at the end of this chapter.

First, however, we will visit sky, clouds, and water—the environmental triad of computer game ambience.

Figure 18.1 A serene scene.

Skyboxes

When you are tasked to create a 3D game that offers unrestricted player movement in unlimited vistas, you will need to come up with ways to provide that open, outdoors perception. A technique that works well is to provide a static background sky that contains elements of the sky that we often

Team LRN

take for granted, like clouds, and a color gradient that changes as you move farther away from the horizon. We do this using a construct known as a *skybox*.

A skybox is a cube that surrounds the game player. The player stands *inside* the box, and no matter which way he turns, he will see some part of the box as long as it isn't obscured by other in-game objects. The box never rotates, and the sides are always the same distance away from the player no matter how far or how fast he moves. Because of the way the images on the faces of the skybox are created, the player does not have the feeling that he is inside a big cube. The skybox images are on the inside faces of this cube, as you can see in Figure 18.2. The back view has been left out to help illustrate the point.

When using skyboxes, we treat them as if they are infinitely large. Only objects that the player can never reach will look correct, like clouds in the sky or distant mountains. If you limit a player's movement to just viewing from a fixed location, you could even use a skybox for nearby scenery.

Figure 18.3 shows an exploded view of the skybox images and how they relate to each other. Note that the image for the bottom is a black field. If you were depicting an area with a usable view in that direction, you would of course include an appropriate image.

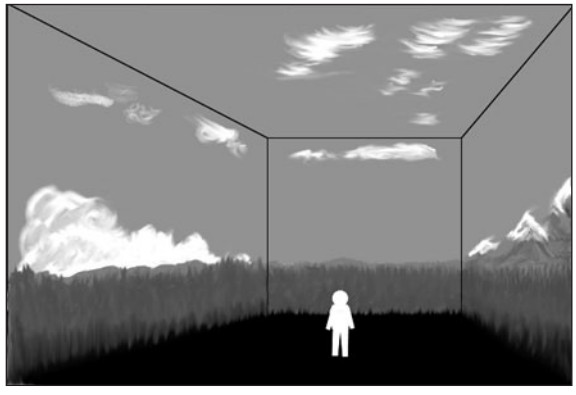


Figure 18.2 A pictorial skybox.

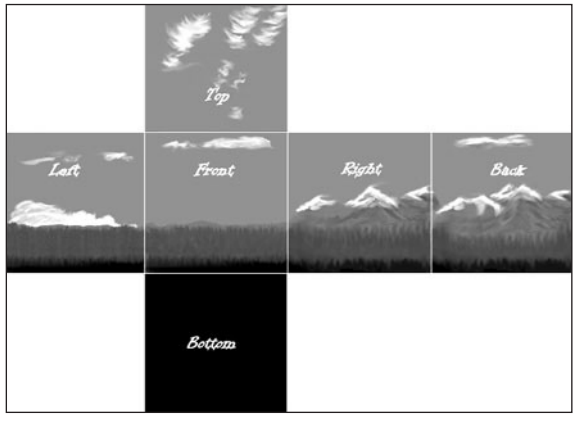


Figure 18.3 An exploded skybox.

To create the illusion that the player is embedded in a large and seamless world, there are two things that you must get right when creating a skybox: seamless, matching adjacent edges and correct perspective.

The edge matching issue is one we are already familiar with from previous texture endeavors.

The perspective issue is a little less obvious when you first consider making skyboxes—but take a look at Figure 18.4. Remember that the skybox is always the same distance away from the player, and the orientation is fixed. The front face, if it happens to be facing north, will always face north, no matter which way the player is facing or looking. This causes a visual problem when viewing the images on the skybox faces.

The image areas that are on the face closest to the player will appear larger than the portions of the image nearest the corners, because the corners are farther away. Figure 18.5 simulates what that would look like.

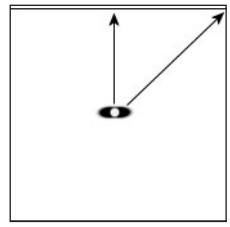


Figure 18.4 Skybox edge distances.

In order to remove the distortion when the image is viewed in game, we need to distort its appearance *outside* the game environment in such a way that when the perspective comes into play in game, the image looks natural. Figure 18.6 shows such a predistorted image.

Each of the six square skybox images should be created with the same resolution. The most common resolution to use is 256 by 256 pixels. The higher the resolution, the better the skybox will look in most cases, but there is a limit beyond which higher skybox image resolution doesn't help the appearance. Because we are always worried about memory used and processing time consumed, we want to make sure we don't go higher than the maximum. If you are interested in using larger skybox images, there is a way to calculate the maximum resolution to use as your upper limit, using this mathematical formula:

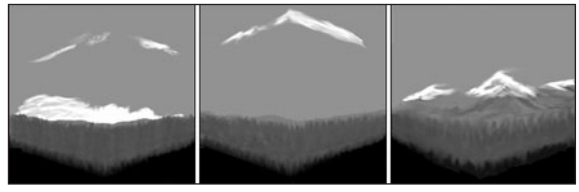


Figure 18.5 Distorted image.

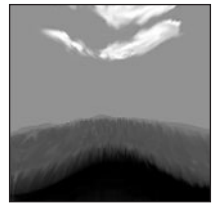


Figure 18.6 Predistorted image.

```
maxSkyboxResX = maxScreenResX * 1/tan(FOV/2)
maxSkyboxResY = maxScreenResY * 1/tan(FOV/2)
```

The basic concept is that the smaller the *Field of View* (FOV), the higher the resolution you will need for the skybox. This is because as the FOV gets smaller, you are looking farther and at a smaller portion of the skybox image. This smaller portion fills the view, and therefore the pixels are larger. Typical first-person point-of-view games use a 90-degree FOV and often have a 60-degree (or even smaller) zoomed-in view for sniper scopes or binoculars.

For example, if our screen resolution is 800 by 600 pixels and our FOV is 90 degrees, then applying our formula yields this:

```
maxSkyboxResX = 800 * 1/tan(90/2)
maxSkyboxResX = 800 * 1/1
maxSkyboxResX = 800
```

It also follows that we don't need to recompute the Y resolution because it will scale proportionally. So for this 800 by 600 pixel display with a 90-degree FOV, the highest resolution we should use for the skybox images is 800 by 600 pixels, by happy coincidence!

However, if we want to know the deal for the 60-degree FOV that our player's binoculars provide, we need to recompute that value as follows:

```
maxSkyboxResX = 800 * 1/tan(60/2)
maxSkyboxResX = 800 * 1/ 0.57735
maxSkyboxResX = 800 * 1.732
maxSkyboxResX = 1386
```

For the Y resolution, the value is 1,039. So if you decide to create a high-resolution skybox, you should probably go with nothing larger than 1,280 by 1,024 pixels. (Most games, including Torque, need the image resolution values to be powers of two.)

Personally, I would go with 1,024 by 1,024 as a reasonable compromise for a maximum resolution. These dimensions would apply to all of your skybox panels in a given skybox. The size you eventually choose for your game will in the end be a judgment call, but by using the previous calculations, not just a hopeful stab in the dark.

Creating the Skybox Images

As with other texture-related issues, there is always the question of where to obtain source material. Once again, you have the option of creating your own by using a digital camera or a camera and scanner combination or by simply drawing your own images.

In this section I will walk you through drawing some clouds on the horizon for your skybox—this is a common sunny-day sort of scene. Low cumulus clouds in the distance peek just above the horizon, all around you.

1. Open Paint Shop Pro and create a new image by selecting File, New. Fill in the dialog box with 256 for both the height and width of the blank image. Make sure that the color depth is set to 16 million colors (24 bit).
2. Save this blank file as C:\aEmaga6\control\data\maps\front.png.
3. Select the Fill tool.
4. Over in the Materials palette, select the Foreground and Stroke Properties check box, just to the right of the color picker (see Figure 18.7).

Figure 18.7 Foreground and Stroke Properties check box.

5. When the Materials dialog box opens, click the Gradient tab.
6. Make sure the value in the Angle window is set to 0, and then click the Edit button.
7. Set the Gradient scale to have two color markers (they look like little pens) on the bottom side of the scale to approximate the settings shown in Figure 18.8. Do the same for the range modifier (the little diamond on the top).

8. Click the leftmost color marker, and then click in the color picker window to its left.
9. In the Color dialog box, enter the RGB values as 215, 215, 255, respectively.
10. In the same manner, set the right-hand marker to 0,0,192.
11. Click OK and then OK again to close the windows and accept the settings.
12. Now click the Fill tool in the image to get a gradient like that shown in Figure 18.9.
13. Change the Foreground and Stroke Properties check box out of gradient mode and into color mode. Then use the Eyedropper to set its color to pure white.
14. Next select the Air Brush tool and set the size to 8, the Hardness and Density to 70, and the Opacity to 15.
15. Now draw some cloudlike shapes between about half and two-thirds of the way from the top of the image so that you get something like Figure 18.10.
16. Create three more versions of this image, naming the others "left.png", "right.png", and "back.png" in the same place you saved front.png. Go ahead and make each one different in its own way, if you like.
17. Make a fifth image that is solid blue, with RGB values of 0,0,192. This color matches the darkest blue in the gradient we made. Name this file "top.png".
18. Make the sixth and final image and fill it in with black. Name this one "bottom.png".

    Now it's time to test out your images.
19. Locate the file in your Emaga6 map folder called C:\aEmaga6\control\data\maps\sky_day.dml. Make a copy of this file in the same directory and name the copy "sky_book.dml".

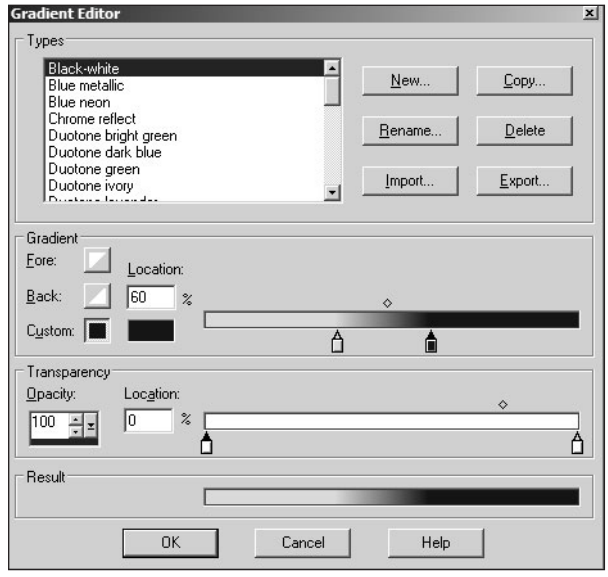


Figure 18.8 The Gradient scale.

Figure 18.9 Image with gradient.

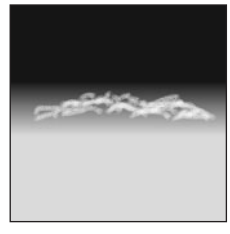


Figure 18.10 Some clouds.

20. Open the sky_book-.dml file with UltraEdit. Change the first six lines to read as follows:

```
skyfront
skyright
skyback
skyleft
skytop
skybottom
```

21. Save the file.
22. Open C:\aEmaga6\control\data\maps\book_ch6.mis and locate the line that looks like this:

```
materialList = "./sky_day.dml";
```

    and replace it with this:

```
materialList = "./sky_book.dml";
```

23. Save the file.
24. Launch the Emaga6 sample program and enter into the game. Take a look around. Notice the corners? See how your clouds become distorted?

You already know how to fix up the textures so that they join seamlessly, so I'll leave you to do that. Note that you *probably* don't have to worry about the top edges, because the top image and the top edges of the side images all have the same RGB value—0,0,192.

Also, the bottom doesn't need to be blended either, because it's not going to be visible beneath our terrain. So that just leaves the perspective distortion to fix.

Adjusting for Perspective

Although we are going to be adjusting for perspective distortion, we aren't going to use the built-in perspective tools in Paint Shop Pro. Instead, we will use the Warp tool.

1. Open up one of your side images, like the front one, for example.
2. Choose Effects, Distortion Effects, Warp. Your image will be distorted as shown in Figure 18.11.
3. Repeat the warping for all three of the other image files so that you've corrected all of the lateral view images, left, right, front, and back.
4. Run Emaga6 and check your work.

Now you might find that after you've done the distortion you now have seams again in your skybox. If so, go back and fix the edges.

There you have it! Your very own do-it-yourself skybox!

Team LRN

The Sky Mission Object

You probably noticed when you were editing the Emaga6 MIS file that there was an object defined in there called "Sky". There are lots of goodies in that object for us sky worshipers.

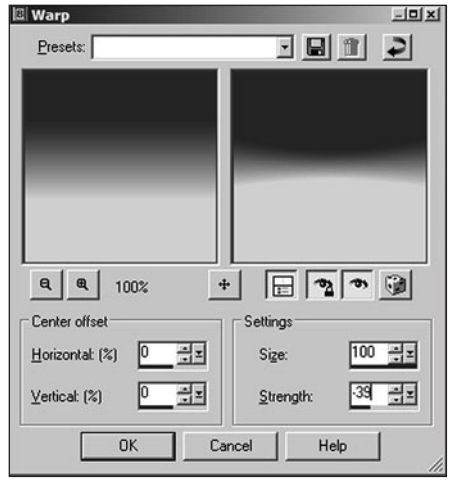


Figure 18.11 Applying perspective-correcting warp.

Here it is:

```
new Sky(Sky) {
   position = "-1088 -928 0";
   rotation = "1 0 0 0";
   scale = "1 1 1";
   materialList = "./sky_book.dml";
   cloudHeightPer[0] = "0.349971";
   cloudHeightPer[1] = "0.25";
   cloudHeightPer[2] = "0.199973";
   cloudSpeed1 = "0.0002";
   cloudSpeed2 = "0.0004";
   cloudSpeed3 = "0.0006";
   visibleDistance = "1100";
   fogDistance = "1000";
   fogColor = "0.820000 0.828000 0.844000 1.000000";
   fogStorm1 = "0";
   fogStorm2 = "0";
   fogStorm3 = "0";
   fogVolume1 = "500 0 100";
   fogVolume2 = "0 0 0";
   fogVolume3 = "0 0 0";
   windVelocity = "0.1 0.1 0";
   windEffectPrecipitation = "1";
   SkySolidColor = "0.547000 0.641000 0.789000 0.000000";
   useSkyTextures = "1";
   renderBottomTexture = "0";
   noRenderBans = "0";
      locked = "true";
};
```

We have already encountered the `MaterialList` property and have seen that it is used to point to a file that contains the names of the images that will be displayed on the interior faces of our skybox.

Not all of the properties in the skybox are particularly interesting; they owe their presence to Torque's beginnings as the code that drives the *Tribes 2* game. The `position`, `scale`, and

`rotation` properties don't accomplish much when you use them—they are there because all objects have those properties whether or not they are meaningful.

The `cloudHeight` properties *are* useful, and we will cover them in the next section. The same applies to the properties for fog.

One of the most useful properties is `visibleDistance`. This property specifies the distance, in world units, beyond which the terrain and all game objects will not be rendered. This is a useful, though rather ham-handed, method for increasing frame rates in game worlds that have many objects present. In conjunction with the `fogDistance` property, this *sort of* simulates the concept that all landscape artists are familiar with that objects become hazier and harder to see at a distance. This is because there is simply more atmosphere between you and the objects you are viewing in the distance, and the greater the distance, the more the air obscures your view. This effect is a well-known one called *atmospheric perspective*. The great Leonardo Da Vinci studied this effect quite a bit back in the 15th and 16th centuries—he called it *aerial perspective*.

By exaggerating this effect we have a useful mechanism to reduce the number of objects that the video card needs to render, and this improves your frame rate.

The `fogDistance` property specifies the distance from you that the haziness we just talked about begins. The distant fogginess starts at this point and gets thicker as the distance increases, until the `visibleDistance` is reached, after which nothing is rendered. By using these two properties, you can prepare a game world where there is a natural-appearing haziness that slowly obscures distant objects until they disappear completely.

note

You should *always* make sure that `visibleDistance` is a bigger number than `fogDistance`, otherwise you risk crashing the game engine in clients in certain situations. In fact, for the sake of safety, always make sure that `visibleDistance` is at least 50 units larger than `fogDistance`. Less than that is not really useful anyway.

If you don't want to use a skybox, there is the `SkySolidColor` property, which you can set. Then you will get a uniformly colored sky all around with a band of changing color near the horizon to simulate the lightening effect we see—something like the gradient we made for our skybox. In this case, to disable the skybox, set `useSkyTextures` to 0 or `False`. Set `noRenderBans` to 0 or `False` to enable the simulated horizon coloring, and set it to `True` to disable the coloring.

You can also just prevent the bottom image in the skybox from being rendered, or considered for rendering, by setting `renderBottomTexture` to 0 or `False`. This might eke out a frame or two of frame rate for you.

The `windVelocity` and `windEffectPrecipitation` precipitation properties have no effect on their own. They are used in conjunction with the storm effects we will cover later.

Cloud Layers

Your game's sky doesn't start and end with the skybox. A beautiful background sky is nice, and important in some settings, but it's static. If you go outside on a nice day and look around, you will often see a sky with clouds that presents itself much like the one you can make with the skybox.

But more often, you will see that *and* you will have clouds moving across the sky above you, blowing in the wind. In fact, you will probably notice layers of clouds—often two layers and sometimes even three layers. The lower layers whip across the view above you, while the upper layers move at a more sedate pace, sometimes even in a different direction.

In Torque we can define up to three layers of moving clouds with the Sky mission object in the MIS file that the server uses to define the game world.

Cloud Specifications

For each layer, we define its altitude as a percentage of a pseudo-altitude. Now this is tricky and might be a bit difficult to understand. The first thing to get is that your player can *never* go up—either in camera fly mode or in a flying vehicle—high enough to reach the lowest cloud layer. In this sense, cloud layers operate somewhat like a skybox. However, you *can* position the three layers relative to each other. The reason for this is so that the motions of each cloud layer can be calculated in correct proportion to each other. If you have a steady wind that is the same at all altitudes, then the lowest cloud layer will seem to move faster than the others, and the highest will seem to move slower than the others. How much faster or slower depends on the distance between the cloud layers and their distance from your player, as the observer.

And that's what the `cloudHeightPer` properties do—they inform the *visual* appearance of the clouds but not their *physical location* in the game.

Now another consideration is that wind speeds are not the same at all altitudes in real life. Usually, the *winds aloft* (winds at altitudes of 1,000 feet or greater above the ground) are higher the higher you go, up to about 30,000 or 40,000 feet or so. Then it starts to get really weird.

You can plug in the movement speeds for the clouds at different altitudes using the `cloudSpeed`*`n`* for each specified `cloudHeightPer[`*`n`*`]` and have the game engine figure out the relative motion based on pseudo-altitude and the speed at that altitude. Unfortunately, Torque doesn't handle wind direction for clouds as well—that would be the final link needed to provide really neat cloud motion. Wind direction is specified by a single `windVelocity` property that applies to all layers. In real life, wind directions *back* and *veer* according to altitude, but we can't do that here.

Team LRN

Using the `windVelocity` property requires a little thought. The value is expressed as an XYZ coordinate. The third value, the Z, is irrelevant, but the X and Y values are used to calculate the vector on the horizontal plane in two dimensions. The vector then points to the world origin (or center). If we look up at the sky and imagine the X- and Y-axes pasted up there, somewhat like in Figure 18.12, we can figure out the direction.

The value "1 1 0" would describe a wind from the southeast, as illustrated in Figure 18.12, and "1 0 0" would be a wind from the east.

Cloud Textures

Now, you need to tell the engine what all those clouds you have zipping around up there actually *look* like. You do this by specifying an image file in the same material file that you used to specify the skybox image files. After the first six lines in that file that indicate the skybox images, the next lines indicate the cloud image files. One line equals one cloud layer, with the first line after the skybox image lines indicating layer one, the next being layer two, and the last being layer three, like this:

```
skyfront
skyright
skyback
skyleft
skytop
skybottom
no_cloud
cloud2
cloud3
```

That is the contents of sky_book.dml. Notice the use of the name "no_cloud" for the first cloud layer. In this example I didn't want to have any clouds at that first layer, so for this layer I created an image file that has no clouds in it.

So, you are asking, how do we make a cloud texture that *does* have clouds? Glad you asked! Let's make one.

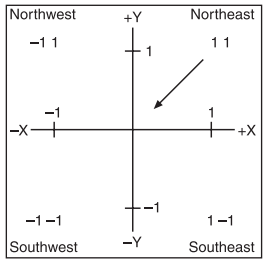


Figure 18.12 Wind velocity conversion correcting warp.

1. Launch Paint Shop Pro and create a new file that is 256 pixels by 256 pixels in size, has a transparent background, and includes 16 million (24-bit) colors.
2. Save this empty image file as C:\aEmaga6\control\data\maps\no_cloud.png.
3. Select the Air Brush tool, set it to spray white, and then spray a little bit around your new image, avoiding the edges, like in Figure 18.13. Or spray the edges, but make sure you adjust the edges so that the image is suitable for tiling.

4. Save your image as C:\aEmaga6\control\data\maps\mycloud.png. (Or you can save it as a JPG, if you like.)
5. Edit C:\aEmaga6\control\data\maps\sky_book.dml so that the last three lines look like this:

```
mycloud
no_cloud
no_cloud
```

Now run your game, and check out the clouds! Of course, you can add more cloud images for the other layers, or you can use the same image for all three.

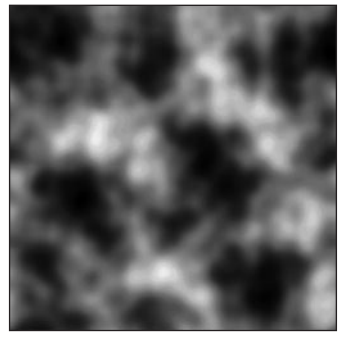


Figure 18.13 A simple cloud texture.

Fog

Fog is, of course, another nifty weather feature. We've already encountered one kind of fog that is used to obscure distant objects and terrains. There is another kind of fog that operates in layers, just like the clouds—except that with fog, these are real layers in the game world that you can actually get into with your player, depending on where the layers are placed.

This layered fog is a limited form of *volumetric* fog. It is limited in the sense that although you can specify the upper and lower bounds of the fog, it will appear at those levels throughout the entire map.

You can use this layered fog to complement the moving cloud textures to create clouds (except that the fog will never be at the same altitude as the cloud textures). You can also deposit fog in low-lying river valleys.

A good use for this fog is underwater, helping to reduce visibility there. This reduced visibility results because of silt and other materials that often exist underwater, cutting down your ability to see far.

This layered volume fog is specified in the Sky mission object that we looked at earlier. An entry looks like this:

```
fogVolume1 = "500 0 100";
```

The three parameter numbers are, in order, distance, bottom, and top. Their meanings are shown in Table 18.1.

You already know how to edit the mission file and change the properties of the various mission objects, so go ahead and putz around with the fog values and see how they work.

Table 18.1 Volume Fog Settings

Parameter	Description
`distance`	View distance when in the layer. This works like the distance fog, except that a value of 0 here means there is no fog at all. If you want a really, really close view distance, use 1, not 0.
`bottom`	Bottom of fog layer.
`top`	Top of fog layer.

Storms

Torque has built-in capabilities to generate storms, using lightning, rain, and thunder. It's pretty cool how this is done. You can manually instigate a storm using script code, and there are some functions provided that will automate portions for you.

note

The lightning storm features require the use of sound effects files, but we don't cover those until the next chapter. So you will have to add the appropriate code to make the sound effects work. This will be done with minimal commentary here—just enough to get the thunder sounds working. See Chapters 19 and 20 for a detailed look at sounds.

Setting Up Sound

There's some preparation we need to do at this point before proceeding with the rest of the weather features. We need to get some sound files, images, and supporting code files and put them in the right places for our game, as follows:

1. In the directory C:\3DGPAi1\RESOURCES\CH18 locate the file SettingsScreen.cs and copy it to the directory C:\aEmaga6\control\client\misc\. Then copy SettingsScreen.gui from the same place to C:\aEmaga6\control\client\interfaces\.
2. Copy the following files to the directory C:\aEmaga6\control\data\sound\:

```
C:\aEmaga6\control\data\sound\thunder1.wav
C:\aEmaga6\control\data\sound\thunder2.wav
C:\aEmaga6\control\data\sound\thunder3.wav
C:\aEmaga6\control\data\sound\thunder4.wav
C:\aEmaga6\control\data\sound\buttonOver.wav
C:\aEmaga6\control\data\sound\rain.wav
```

3. Copy the following files *from* the same place:

```
C:\aEmaga6\control\data\sound\lightning.dml
C:\aEmaga6\control\data\sound\lightning1frame1.png
```

```
C:\aEmaga6\control\data\sound\lightning1frame2.png
C:\aEmaga6\control\data\sound\lightning1frame3.png
C:\aEmaga6\control\data\sound\rain.dml
C:\aEmaga6\control\data\sound\rain.png
```

However, this time they go *to* the directory C:\aEmaga6\control\data\maps\.

4. Edit the file C:\aEmaga6\control\client\initialize.cs and locate the following line:

```
Exec("./interfaces/MasterScreen.gui");
```

and after it, add this line:

```
Exec("./interfaces/SetupScreen.gui");
```

5. Next, locate the line:

```
Exec("./misc/MasterScreen.cs");
```

and after it, add this line:

```
Exec("./misc/SetupScreen.cs");
```

6. Edit the file C:\aEmaga6\control\client\default_profile.cs and add the following line near the top:

```
GuiButtonProfile.soundButtonOver = "AudioButtonOver";
```

7. Copy the file C:\3DGPAi1\RESOURCES\CH18\OpenAL32.dll to the directory C:\aEmaga6\.
8. Locate the file C:\aEmaga6\control\client\initialize.cs and add these lines to the top:

```
$pref::Audio::driver = "OpenAL";
$pref::Audio::forceMaxDistanceUpdate = 0;
$pref::Audio::environmentEnabled = 0;
$pref::Audio::masterVolume   = 1.0;
$pref::Audio::channelVolume1 = 1.0;
$pref::Audio::channelVolume2 = 1.0;
$pref::Audio::channelVolume3 = 1.0;
$pref::Audio::channelVolume4 = 1.0;
$pref::Audio::channelVolume5 = 1.0;
$pref::Audio::channelVolume6 = 1.0;
$pref::Audio::channelVolume7 = 1.0;
$pref::Audio::channelVolume8 = 1.0;

$GuiAudioType     = 1;
$SimAudioType     = 2;
$MessageAudioType = 3;
```

```
new AudioDescription(AudioGui)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = false;
   type     = $GuiAudioType;
};

new AudioDescription(AudioMessage)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = false;
   type     = $MessageAudioType;
};

new AudioProfile(AudioButtonOver)
{
   filename = "~/data/sound/buttonOver.wav";
   description = "AudioGui";
       preload = true;
};
```

Now that we've done that, we can move on to the storm-specific stuff.

9. Type the following into a new file and save it as C:\aEmaga6\control\server\misc\weather.cs.

```
datablock AudioProfile(HeavyRainSound)
{
   filename    = "~/data/sound/rain.wav";
   description = AudioLooping2d;
};
datablock AudioProfile(ThunderCrash1Sound)
{
   filename  = "~/data/sound/thunder1.wav";
   description = Audio2d;
};
datablock AudioProfile(ThunderCrash2Sound)
{
   filename  = "~/data/sound/thunder2.wav";
   description = Audio2d;
};
datablock AudioProfile(ThunderCrash3Sound)
```

```
{
   filename  = "~/data/sound/thunder3.wav";
   description = Audio2d;
};
datablock AudioProfile(ThunderCrash4Sound)
{
   filename  = "~/data/sound/thunder4.wav";
   description = Audio2d;
};
datablock LightningData(LightningStorm)
{
   strikeTextures[0]  = "~/data/maps/lightning.dml";
   thunderSounds[0] = ThunderCrash1Sound;
   thunderSounds[1] = ThunderCrash2Sound;
   thunderSounds[2] = ThunderCrash3Sound;
   thunderSounds[3] = ThunderCrash4Sound;
};
datablock PrecipitationData(HeavyRain)
{
   type = 1;
   materialList = "~/data/maps/rain.dml";
   soundProfile = "HeavyRainSound";
   sizeX = 0.1;
   sizeY = 0.1;
   movingBoxPer = 0.35;
   divHeightVal = 1.5;
   sizeBigBox = 1;
   topBoxSpeed = 20;
   frontBoxSpeed = 30;
   topBoxDrawPer = 0.5;
   bottomDrawHeight = 40;
   skipIfPer = -0.3;
   bottomSpeedPer = 1.0;
   frontSpeedPer = 1.5;
   frontRadiusPer = 0.5;
};
```

10. Finally, add some datablocks to the mission file to cause our new storm features to load when the game launches. Locate the mission file again, C:\aEmaga6\control\data\maps\book_ch6.mis, and find the last two lines of code, which should look like this:

```
};
//--- OBJECT WRITE END ---
```

And add the following two datablocks *before* those last two lines:

```
new Precipitation(RainStorm) {
   position = "-45.0071 -29.016 244.517";
   rotation = "1 0 0 0";
   scale = "1 1 1";
   nameTag = "rs";
   dataBlock = "HeavyRain";
   offsetSpeed = "0.25";
   minVelocity = "1.5";
   maxVelocity = "3";
   color1 = "1.000000 1.000000 1.000000 1.000000";
   color2 = "-1.000000 0.000000 0.000000 1.000000";
   color3 = "-1.000000 0.000000 0.000000 1.000000";
   percentage = "1";
   maxNumDrops = "5000";
   MaxRadius = "60";
};
new Lightning(ElectricalStorm) {
   position = "200 100 300";
   rotation = "1 0 0 0";
   scale = "250 400 500";
   datablock = "LightningStorm";
   strikesPerMinute = "30";
   strikeWidth = "2.5";
   chanceToHitTarget = "100";
   strikeRadius = "250";
   boltStartRadius = "20";
   color = "1.000000 1.000000 1.000000 1.000000";
   fadeColor = "0.100000 0.100000 1.000000 1.000000";
   useFog = "1";
   locked = "true";
};
```

That should do it. Launch your game, and enjoy the storm!

Storm Materials

You will have noticed that when you copied those files from the book's resources directory, there were DML (material definition) and PNG files for both the lightning and rain. If you look inside the rain.dml file, you will see this one line:

```
rain.png
```

Figure 18.14 shows what this texture looks like. It has 16 images of raindrops in a 4 by 4 grid arrangement.

Now, the actual texture file has a difference—the areas shown in black in Figure 18.14 are really transparent when viewed in the file. To create your own such file, make a new file in Paint Shop Pro, and set it to 128 pixels square, 16 million colors (24 bits), and transparent using the New Image dialog box. Then choose View, Change Grid Guide and Snap Properties, and set the vertical and horizontal values in the Current Image Settings to 32 for both. A 4 by 4 grid will appear in your view of the new blank image. Draw your version of each of the 16 drops in a grid box on the image. The grid is not part of the image. Save the file and deposit it in the same place where you had put the rain.png file. Then edit the rain.dml file to point to your new version instead of the original.

The same process applies to the lightning images, except that the lightning.dml material file looks like this:

```
lightning1frame1
lightning1frame2
lightning1frame3
```

The lightning.dml material file is a list of lightning image files that are displayed in sequence as the lightning stroke occurs. Figure 18.15 shows each of these images in order, from left to right.

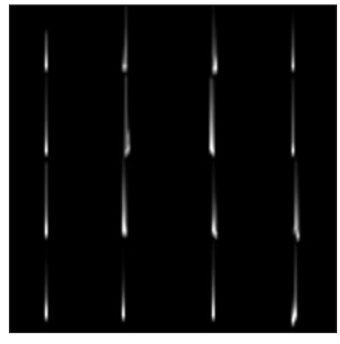


Figure 18.14 Raindrop images.

When making the lightning frame files, you need to make them 128 pixels wide by 256 pixels high. Draw your lightning bolts on a black background—all the areas you leave black will be treated as transparent. That is, *they really are black* and are not just rendered that way for purposes of the picture, as was the case back with Figure 18.14.

Lightning

Now, let's take a look at what makes lightning tick, as it were. There are two significant declarations: one is the `LightningData` datablock in the server code, and the other is the `Lightning` object definition that resides in the mission file. The datablock is transmitted to the client when the mission is loaded with the `Lightning` object definition

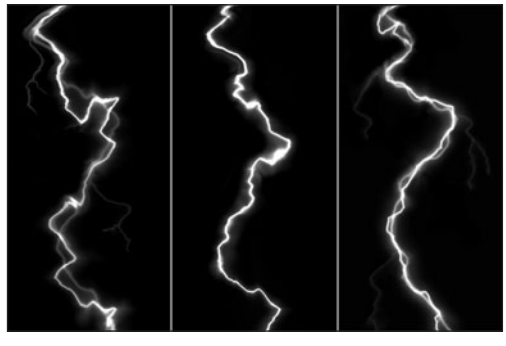


Figure 18.15 Lightning stroke images.

getting transmitted to the client. The datablock describes what resources are used to create the lightning visuals and sound effects, as follows:

```
datablock LightningData(LightningStorm)
{
   strikeTextures[0]  = "~/data/maps/lightning.dml";
   thunderSounds[0] = ThunderCrash1Sound;
   thunderSounds[1] = ThunderCrash2Sound;
   thunderSounds[2] = ThunderCrash3Sound;
   thunderSounds[3] = ThunderCrash4Sound;
};
```

Every time Torque triggers the thunder, one of the listed `thunderSound`*n* properties is chosen randomly.

The `Lightning` object defines how the lightning actually works in the game, as follows:

```
   new Lightning(ElectricalStorm) {
      position = "200 100 300";
      rotation = "1 0 0 0";
      scale = "250 400 500";
      datablock = "LightningStorm";
      strikesPerMinute = "30";
      strikeWidth = "2.5";
      chanceToHitTarget = "100";
      strikeRadius = "250";
      boltStartRadius = "20";
      color = "1.000000 1.000000 1.000000 1.000000";
      fadeColor = "0.100000 0.100000 1.000000 1.000000";
      useFog = "1";
   };
```

Obviously, it's important to indicate which datablock to use. This is done with the `datablock` property. There are then a couple of self-evident properties: `strikesPerMinute` and `chanceToHitTarget`. Then `strikeWidth` indicates the scale factor applied to the image overlay of the lightning bolt that comes from the image files.

When a bolt is generated, a random spot within a circular area is chosen to be the place where the bolt begins, and then another random spot within a different circular area is chosen to be the spot where the bolt hits. The size of the starting area is defined by `boltStartRadius`, and the size of the strike area is defined by `strikeRadius`.

The centers of the start and strike areas are defined by the `position` property. The whole shebang can be made larger or smaller based on the `scale` property. The `rotation` property has no effect.

The `color` property defines a coloring that is applied when the bolt first appears, and the color values are changed over the life of the bolt until they reach the settings in `fadeColor`.

The `useFog` property indicates whether the fog defined by the `stormFog`*n* property in the Sky mission object will be used.

In Figure 18.16 you can see a lighting bolt coming out of the sky in the game setting.

Rain

You can make it rain in much the same way as you make thunder and lightning, though there are differences in the details.

For one thing, the `Precipitation` datablock is much bigger.

```
datablock PrecipitationData(HeavyRain)
{
   type = 1;
   materialList = "~/data/maps/rain.dml";
   soundProfile = "HeavyRainSound";
   sizeX = 0.1;
   sizeY = 0.1;
   movingBoxPer = 0.35;
   divHeightVal = 1.5;
   sizeBigBox = 1;
   topBoxSpeed = 20;
   frontBoxSpeed = 30;
   topBoxDrawPer = 0.5;
   bottomDrawHeight = 40;
   skipIfPer = -0.3;
   bottomSpeedPer = 1.0;
   frontSpeedPer = 1.5;
   frontRadiusPer = 0.5;
};
```

Significant properties here are `sizeX` and `sizeY`, which dictate the scaled size of the drops.

The rest of the properties are not well documented and are hard to decipher. Of course, you are free to fiddle with them to your heart's content. The settings included in the preceding code work well.

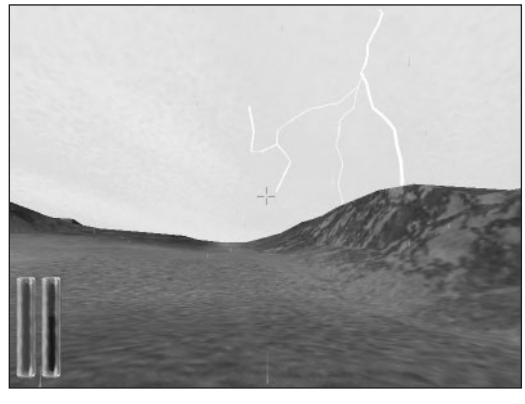


Figure 18.16 A lightning bolt.

If you do experiment with the datablock, realize that the rain is generated at the client. Other players don't see the exact same raindrops at the same instant that you do—it would be lunacy to try to make the server track each drop! Therefore, the rain is generated in a box that envelops the player and moves around as the player moves. The drops are initiated at the top of the box and tracked and rendered as they move down, pulled by gravity. That little bit of detail might help you with your experimentation.

The `Precipitation` object declaration looks like this:

```
new Precipitation(RainStorm) {
       position = "-45.0071 -29.016 244.517";
       rotation = "1 0 0 0";
       scale = "1 1 1";
       dataBlock = "HeavyRain";
       offsetSpeed = "0.25";
       minVelocity = "1.5";
       maxVelocity = "3";
       percentage = "1";
       maxNumDrops = "5000";
       MaxRadius = "60";
   };
```

The significant properties here are `offsetSpeed`, which describes how quickly the rain moves across in front of you, and `minVelocity` and `maxVelocity`, which describe the lower and upper bounds (respectively) of randomly chosen drop velocities. The latter two keep the rain from falling as a uniform sheet, thus giving it a more realistic appearance.

The `percentage` property dictates the density of the drops, while `maxNumDrops` indicates the upper bounds of the number of drops to create. `MaxRadius` is the outer bounds of the area surrounding the player where the engine can create drops.

A Perfect Storm

Okay, so it may not be perfect, but it is neat. There are two useful object methods that you can use to move a storm in and out gradually without the need to manipulate the storm-related objects in detail.

The first is the `stormCloud` method that belongs to the `Sky` object. It looks like this:

```
Sky.stormCloud (flag, fade)
```

Set the flag to 1 if you want to create storm clouds and 0 if you want them to go away. To use the method, you would first call `Sky.stormCloud (0, 0)` as soon as your game starts to ensure that the clouds are not visible—all you would see is the skybox.

Then, at the moment you decide to call up a storm with your wand, you would call `Sky.stormCloud (1, 60)` somewhere in your script. This will cause the engine to gradually fade in your clouds over a 60-second time frame. When the storm clears, you make them go away gradually by calling `Sky.stormCloud (0, 60)`. Of course, you could use a different fade value, making it as long or short as you desire.

The second method is a nice complement to the `stormCloud` method. It is called `stormPrecipitation` and belongs to the `Precipitation` object. It looks like this:

```
Precipitation.stormPrecipitation(flag, fade)
```

It works the same way as `stormCloud` but obviously applies to the precipitation.

Use the two methods together, with appropriate fade values, to get a nice storm effect. Try them out in your sample game by entering the commands manually in the console.

Water Blocks

Water blocks are special objects that we can insert into our game world via a mission file. Here is a water block:

```
new WaterBlock(Water) {
   position = "-1024 -1024 0";
   rotation = "1 0 0 0";
   scale = "2048 2048 125";
   UseDepthMask = "1";
   surfaceTexture = "./water";
   liquidType = "OceanWater";
   density = "1";
   viscosity = "15";
   waveMagnitude = "1";
   surfaceOpacity = "0.75";
   envMapIntensity = "0.7";
   TessSurface = "50";
   TessShore = "60";
   SurfaceParallax = "0.5";
   FlowAngle = "0";
   FlowRate = "0";
   DistortGridScale = "0.1";
   DistortMag = "0.05";
   DistortTime = "0.5";
   ShoreDepth = "20";
   DepthGradient = "1";
   MinAlpha = "0.03";
```

```
    MaxAlpha = "1";
    removeWetEdges = "0";
    specularColor = "1.000000 1.000000 1.000000 1.000000";
    specularPower = "6";
    params0 = "0.32 -0.67 0.066 0.5";
    extent = "100 100 10";
    textureSize = "32 32";
    params2 = "0.39 0.39 0.2 0.133";
    envMapOverTexture = "./day_0007";
    params3 = "1.21 -0.61 0.13 -0.33";
    params1 = "0.63 -2.41 0.33 0.21";
    seedPoints = "0 0 1 0 1 1 0 1";
    floodFill = "1";
};
```

Water blocks repeat in the same way that terrain blocks repeat, and because water blocks are flat, the only positioning information of real interest is the height. Table 18.2 describes the significant properties (and there are many!).

Water block textures, as described in various places in Table 18.2, can be created in exactly the same way as cloud textures. In fact, you can even get away with using cloud textures in a pinch!

Take a look at Figure 18.17 to see a water block in action.

Terraforming

You've already seen in Chapter 12 how to create a terrain using height maps. Torque also has a built-in Terrain Editor that you can use to manually modify the terrain height map and square properties.

Figure 18.17 Water in a game setting.

Terrain editing is done using a Terrain brush. The brush is a selection of terrain points centered on the mouse cursor in either a circular or square configuration of different selectable sizes, as you can see in Figure 18.18.

The brush can also be either a hard brush that has a uniform effect across the surface of the brush or a soft brush whose influence on terrain diminishes toward the edges of the brush. You can adjust the soft

Table 18.2 Water Block Properties

Property	Description
surfaceTexture	Specifies the texture generally used for the surface.
ShoreTexture	Specifies the texture used in shallow areas.
envMapOverTexture	Defines the environment map texture used when looking over the fluid surface.
envMapUnderTexture	Defines the environment map texture used when looking under the fluid surface.
surfaceOpacity	Specifies the maximum opacity of the surface (0.0 -> 1.0).
envMapIntensity	Specifies the intensity of the applied environment map (0.0 -> 1.0). Setting the intensity to 0 results in the environment map pass being skipped, which increases performance slightly.
UseDepthMask	Toggles the depth map feature on and off.
ShoreDepth	Specifies the depth at which the shore texture will start being applied. Larger values result in larger shore texture areas.
DepthGradient	Specifies the gradient that the shore textures will interpolate between MinAlpha and MaxAlpha. The value of 1 equates to linear interpolation, whereas values 0 -> 1 equate to fast fade-out/slow fade-in and the values 1 -> inf equate to slow fade-out/fast fade-in (from deep to shallow).
MinAlpha/MaxAlpha	Specifies the alpha levels used from shore to deep fluid. The MinAlpha can be used to prevent totally transparent areas. You will always be able to see underneath the fluid surface, so use the fog volumes from the Sky object to restrict visibility underwater.
TessSurface/TessShore	Specifies the number of times the textures are repeated over the water block surface for surface/shore textures.
SurfaceParallax	Renders the surface as two layers. When the surface is distorting or flowing, then this controls the ratio of one surface with respect to the other. If you set this to 0.5, then one surface will move at half the speed of the other.
FlowAngle/FlowRate	Specifies the way the fluid flows. The FlowRate controls how fast the fluid flows, and the FlowAngle is a polar angle controlling its direction. Using a FlowRate of 0 completely stops the fluid from flowing.
DistortGridScale/ DistortMag/DistortTime	Controls the distortion effect of the fluid surface. This allows you to create many different surfaces. To control the speed, use DistortTime. Use DistortMag to control the overall magnitude of the distortion. DistortGridScale normally does not need adjusting but can be used to adjust a setting for a small water block that may not look correct on a large one.

Figure 18.18 Terrain brush.

brush fall-off rate in the Terrain Editor Settings dialog box, found under the menu Edit, Terrain Editor Settings.

1. Run your sample game, and when you have spawned into the game, press F8 to switch to fly mode.
2. Fly up a fair bit to get a good overview of the surrounding terrain.
3. Press F11 to switch to the Editor GUI.
4. Choose Window, Terrain Editor.
5. Wave your cursor over the terrain, and notice the Terrain brush marked on the terrain.
6. Drag your mouse up and down after clicking over some area of terrain. You will see your terrain change to conform.
7. Experiment with using different actions to see how the Terrain Editor works.

tip

Every now and then while in the Terrain Editor, press Alt+L to redo the lighting. The cursor will freeze for a few moments while the lighting is done. This will cause the new terrain changes you've made to properly generate shadows.

8. Every now and then remember to save your work.

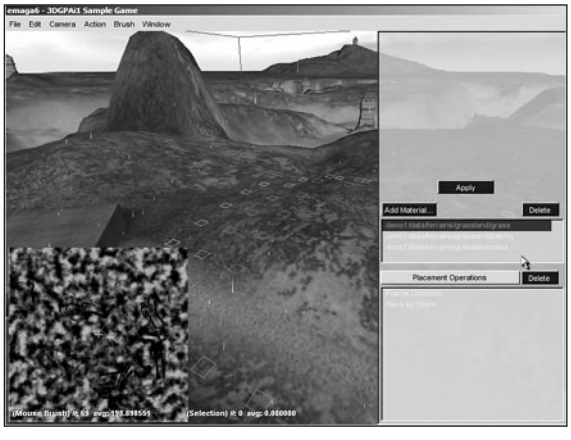


Figure 18.19 Terrain Terraform Editor.

Table 18.3 describes the Terrain Editor functions that are available in the Action menu.

Table 18.4 describes the functions of the Terrain Terraform Editor (see also Figure 18.19), the one we used in Chapter 12 dealing with height maps.

Table 18.3 Terrain Editor Functions

Function	Description
`Select`	Selects grid points that will be painted with the brush.
`Adjust Selection`	Raises or lowers the currently selected grid points as a group.
`Add Dirt`	Adds "dirt" to the center of the brush.
`Excavate`	Removes "dirt" from the center of the brush.
`Adjust Height`	Drags the brush selection to raise or lower it.
`Flatten`	Sets the area bounded by the brush surface to be a flat plane.
`Smooth`	Smoothes, within the bounds of the brush, rough areas of varying terrain height.
`Set Height`	Sets the terrain within the brush to a constant height as specified in the Terrain Editor settings.
`Set Empty`	Converts the squares covered by the brush into holes in the terrain.
`Clear Empty`	Makes the squares covered by the brush solid.
`Paint Material`	Paints the current terrain texture material with the brush.

Table 18.4 Terrain Terraform Editor Functions

Function	Description
`fBm Fractal`	Creates bumpy hills.
`Rigid Multifractal`	Creates ridges and sweeping valleys.
`Canyon Fractal`	Creates vertical canyon ridges.
`Sinus`	Creates overlapping sine wave patterns with different frequencies useful for making rolling hills.
`Bitmap`	Imports an existing 256 by 256 bitmap as a heightfield.
`Turbulence`	Perturbs another operation on the stack.
`Smoothing`	Smoothes another operation on the stack.
`Smooth Water`	Smoothes water.
`Smooth Ridges/Valleys`	Smoothes an existing operation on edge boundaries.
`Filter`	Filters an existing operation based on a curve.
`Thermal Erosion`	Erodes an existing operation using a thermal erosion algorithm.
`Hydraulic Erosion`	Erodes an existing operation using a hydraulic erosion algorithm.
`Blend`	Blends two existing operations according to a scale factor and mathematical operator.
`Terrain File`	Loads an existing terrain file onto the stack.

Team LRN

Moving Right Along

So, you've now seen how you can create and modify your game environment. The three main environmental elements are Sky, Clouds, and Water. We looked at the different ways each of those three elements can be created using tools and techniques available in Torque.

In most cases, you will probably use some form of all of those techniques when you create your game. For example, you would judiciously mix overhead cloud layers with skybox renderings of distant clouds on the horizon.

We've also looked at the combined weather effects involved in storms, and how you can initiate an automated process to start and end storms over time using Torque Script.

In this chapter, we were introduced to sounds in the form of thunder for the lightning strikes. In the next chapter we will more thoroughly explore how to incorporate sounds in our game.

CHAPTER 19

Creating and Programming Sound

As I mentioned in Chapter 1, audio artists compose the music and sound in a game. Good designers work with creative and inspired audio artists to create musical compositions that intensify the game experience.

It also bears repeating that audio artists work closely with the game designers determining where the sound effects are needed and what the character of the sounds should be. They often spend quite a bit of time experimenting with sound-effect sources, looking for different ways to generate the precise sound needed. Visit an audio artist at work and you might catch him slapping rulers and dropping boxes in front of a microphone. After capturing the basic sound, an audio artist will then massage the sound with sound-editing tools, varying the pitch, speeding up the sound or slowing it down, removing unwanted noise, and so on. It's often a tightrope walk balancing realistic sounds with the need sometimes to exaggerate certain characteristics in order to make the right point in the game context.

When creating your game, you have a choice between two basic approaches: obtain a good source of sound effects and music (like an audio library) or create your own sounds. Of course, you also have the option to combine the two approaches. Audio libraries are available from a wide variety of sources, and the commercial ones are quite thorough and professionally made. There are audio libraries available via the Internet for free, but the quality of these sources varies widely in breadth, depth, and recording fidelity.

In this book we are going to take the do-it-yourself approach. The main advantage of going this way is the price; a secondary advantage is that you have total control over the contents of your sound files.

Team LRN

Audacity

There are several tools available to use for recording and editing sound effects and music. A very good open source program—it doesn't cost you anything to use and is made available under the GNU General Public License—is Audacity.

This chapter will show you how to use Audacity (see Figure 19.1) to make sounds for use in your game.

Installing Audacity

To install Audacity, do the following:

1. Browse to your CD in the \Audacity directory.
2. Locate the audacity-win-1_0_0.exe file and double-click it to run it.
3. Click the Next button for the Welcome screen.
4. Follow the various screens, and take the default options for each one, unless you know you have a specific reason to do otherwise.

Using Audacity

You need to ensure that you've got your microphone set up properly—connected to the MIC or microphone input jack on your sound card. Of course, you don't need to obtain your sounds directly from a microphone; you can record from a CD or another audio source. In any event, you need to have that source connected to the correct input and ensure that your audio mixer is set up to record from that source. You should refer to your sound card documentation if you don't know how to do this.

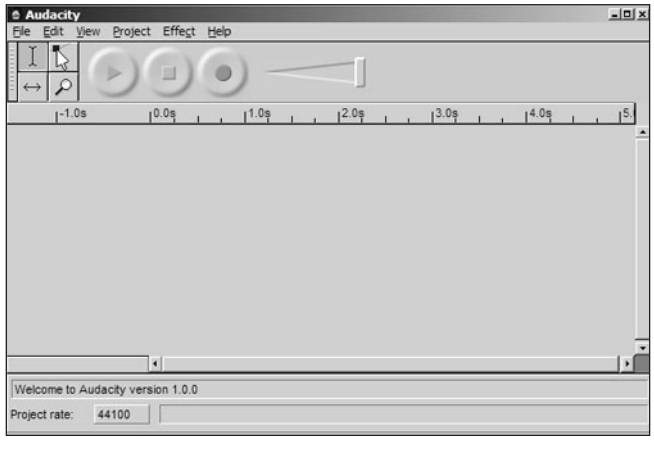


Figure 19.1 Audacity main window.

The basic operation of Audacity is quite straightforward for recording, simple editing, playback, and saving your data.

Recording

Let's record some sound:

1. Launch Audacity by choosing Start, Programs, Audacity, Audacity. You will get the main window, as you saw earlier in Figure 19.1.
2. Click the Record button, as shown in Figure 19.2.

The program is now recording from the microphone. You can see the progress of the recording and the waveforms of the sounds in the window as the recording proceeds, as shown in Figure 19.3.

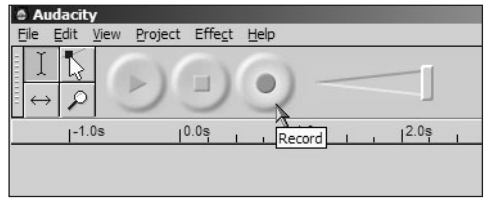


Figure 19.2 The Record button.

3. Speak into the microphone, or if you don't want to hear your own voice, make a noise, like slapping a book down on the desk or something. You will see the sound you made appear in the waveform. Figure 19.4 shows the waveform created when I tapped a pen on the desk next to the microphone.

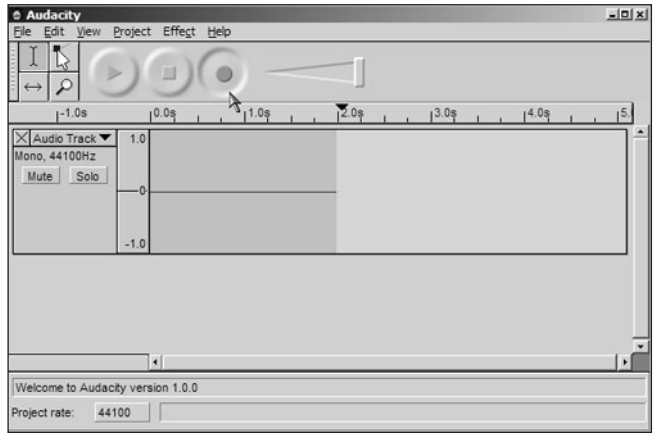


Figure 19.3 Recording in progress.

4. When you have your sound recorded, click the Stop button, as shown in Figure 19.5.
5. Now you can play back your recording, by clicking the Play button, as shown in Figure 19.6.

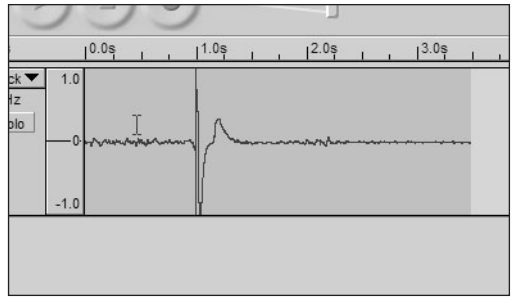


Figure 19.4 Waveform in Audacity.

We'll continue working with Audacity in a moment, but first I want to point out that if you have a waveform but don't hear any sound, make sure that you have the volume turned up high enough in your speakers. Also be sure that it is turned up high enough—and is not muted—in your Windows Volume Control applet (in the Control Panel, and usually on the Windows System Tray on the right side of the taskbar).

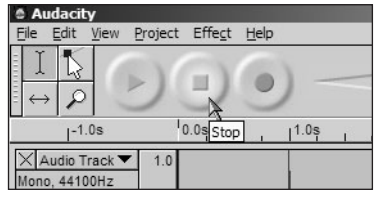


Figure 19.5 Stop recording.

Simple Editing

Now if you're like me you probably have a long period of dead air before the sound effect you made and another chunk afterward. That's fine, because it's easy to fix. So, picking up where we left off in the previous section:

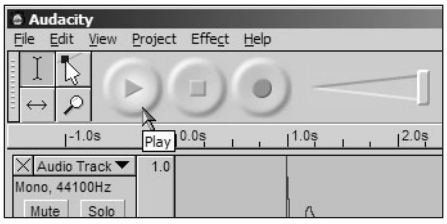


Figure 19.6 Playback.

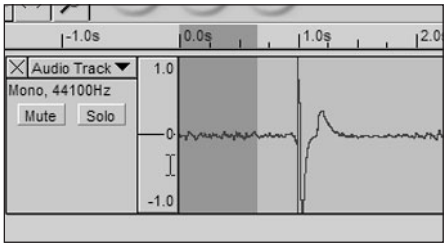


Figure 19.7 Selecting a portion of the waveform.

6. Place your cursor to one side of the portion of the waveform you want to eliminate, and drag it across to the other side. This selects an area to be worked on. See Figure 19.7.
7. Choose Edit, Delete. The selected portion will be excised from the waveform.
8. Repeat the preceding two steps for the unwanted portions of the waveform on the other side of your sound effect. Eventually you will end up with something like Figure 19.8.

We're not finished with our procedure yet; there's still some exporting to do. But before we cover that, I want to mention that above the waveform panel is a series of numbers on a scale. This is the elapsed time scale. The example in Figure 19.8 shows that my final waveform is just a little more than three-quarters of a second in duration.

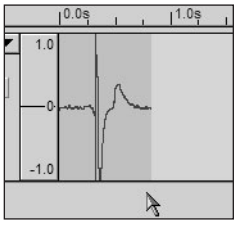


Figure 19.8 The final waveform of the sound effect.

Exporting

Now, once again picking up where we left off, you need to save the sound effect as a file before you can use it:

9. Choose File, Export as WAV. Name your file and save it somewhere convenient for the moment, such as on your desktop.
10. Browse to your desktop (or wherever you saved your file) and double-click your newly created file. Whichever program is set up to play sounds in Windows on your computer will be launched and play your sound.

There are other export options available, but we'll stick with the WAV format for its simplicity and wide availability on Windows platforms. For other platforms, Ogg Vorbis is probably the format of choice on Linux, and AIFF for Macintosh.

Audacity Reference

This section contains some useful reference details to help you use Audacity.

The Main Screen

Figure 19.9 shows the Audacity main screen, with the major components labeled. This section will provide some detail on those components.

The toolbar is where you will find the tools that you will probably use more than any other tools available with Audacity. Use Figure 19.9 to locate the tools in the toolbar, and Table 19.1 to review their functions.

The Track Panel contains tools for managing specific tracks and groups of tracks. See Table 19.2 for details.

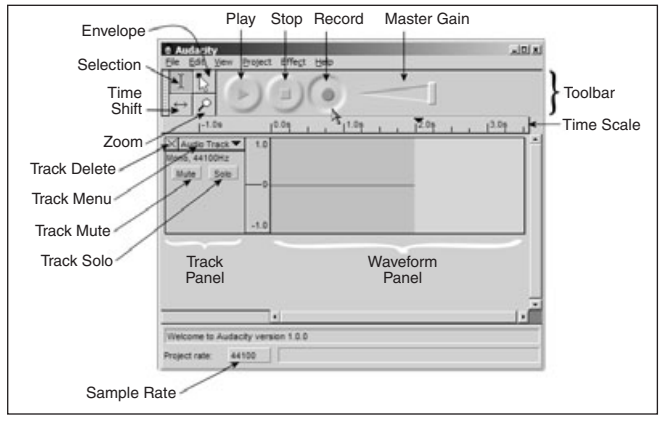


Figure 19.9 The Audacity main screen.

Table 19.1 Toolbar Tools

Tool	Description
Selection	Selects a portion of the audio track. You can set the position of the track cursor simply by clicking at the right place in the track. Select a range of audio by clicking and dragging over the desired portion. Select multiple tracks by dragging across the tracks. Playback begins at the position of the track cursor and will play to the end of the track. If you have selected a range of audio, only the selected range will be played.
Time Shift	Changes the positioning of tracks relative to one another in time. Select this tool, then click in a track and drag it to the left or right.
Envelope	A handy audio processing tool. Its presence directly in the main window of the program is an innovative move. You get detailed control over how tracks fade in and out, right in the main track window with this tool. When you select the Envelope tool, the amplitude envelope of each track is highlighted in a green line; there are control points at the beginning and end of each track. To move a control point, click it and drag it to its new position. To add a new point, click anywhere in the track where a control point doesn't already exist. To remove a point, drag it outside of the track.
Zoom	Zooms in or out of a specific part of the audio. Clicking anywhere in the audio will zoom you in. Right-click or shift-click to zoom out. You can also zoom into a region by dragging the mouse to highlight the region while you have the Zoom tool selected.
Play	Enables you to listen to the audio file currently loaded or to a recording you have just created. The spacebar can be used as a stop and start toggle. Playback always begins at the current cursor position. If a region of audio is selected, only the selected region will play. To play the entire project, choose Edit, Select All and then click the Play button. All tracks on a given channel will be mixed automatically for playback.
Stop	Halts playback.
Record	Records a new track from your microphone or another input device. You can configure recording options by choosing Edit, Preferences. Recording always happens at the project's sample rate.
Master Gain	Controls the volume of the audio output by Audacity to your hardware. Volume increases as you move the slider from left to right.

Team LRN

Table 19.2 Track Panel Tools

Tool	Description
Track Menu	Allows the user to display a track in different formats. This menu also provides the Set Name option that allows the user to create a name for a given track.
Track Delete	Immediately deletes a track, without the option to undo. Use this button carefully.
Solo	Switches the current track to solo mode. You can change a track out of solo mode by clicking it again. When in solo mode, the button for that track turns red. Only tracks that have the Solo button enabled will be played when in solo mode.
Mute	Switches off a track without deleting it. You can unmute a track just by clicking the Mute button again. When muted, a track's Mute button will be green.

Table 19.3 Track Types

Tool	Description
Audio	Audio tracks contain digitally sampled sounds. Two stereo channels are represented by two stereo tracks. Each audio track has a sample rate that is the same as the project sample rate.
Note	Note tracks display data loaded from a MIDI file. They cannot be changed or played, only viewed.
Label	Label tracks can be used to mark a document with annotations. Annotations can be saved to a text file.

Audacity supports three different track types that can be viewed simultaneously when they exist in a single channel. These three track types let you view waveform, MIDI information, and label information for a given audio file. Table 19.3 describes each of the three types.

note

Common values for audio sample rates are shown in Table 19.4.

Table 19.4 Common Sample Rates

Frequency	Usage
8000 Hertz	Typical telephone
11025 Hertz	Minimum "voice quality"
16000 Hertz	Typical "voice quality"
22050 Hertz	Common digital interactive media
44100 Hertz	CD audio, DAT (digital audiotape)
48000 Hertz	Digital studio quality
96000 Hertz	Digital studio quality (newer)

Team LRN

Menus

The Audacity menus provide access to functions for managing files, editing, adjusting views, managing Audacity projects, and finally, creating special effects. There is also a standard Help menu.

The File Menu

Figure 19.10 shows the File menu, and Table 19.5 contains an itemized description of the menu. Note that menu items that have names ending with three ellipsis points (three dots) will bring up a dialog box where you can fill in some parameters.

The Edit Menu

Figure 19.11 shows the Edit menu, and Table 19.6 contains an itemized description of the menu. Parts of this menu contain the standard Cut, Copy, and Paste functions; the rest are related functions that are specific to Audacity's capabilities.

The View Menu

The View menu provides functions that you can use to control what you see in the Audacity window and how you see it. Figure 19.12 shows the View menu, and Table 19.7 contains an itemized description of the menu.

The Project Menu

Audacity uses the concept of projects that you've encountered elsewhere, such as with UltraEdit earlier in this book. By using projects, you can organize data files as well as configuration and operational parameters in one collection that can be recalled at any time. This really helps when dealing with complex tasks. Figure 19.13 shows the Project menu, and Table 19.8 contains an itemized description of the menu.

The Effect Menu

Audacity includes many built-in effects and also lets you use plug-in effects. To apply an effect, simply select part or all of the tracks you want to modify, and select the effect from the menu. Figure 19.14 shows the Effect menu, and Table 19.9 contains an itemized description of the menu.

Figure 19.10 File menu.

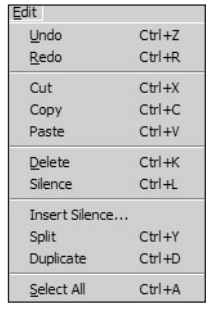


Figure 19.11 Edit menu.

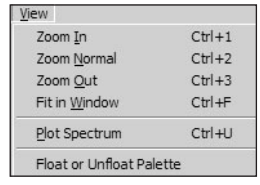


Figure 19.12 View menu.

Figure 19.13 Project menu.

Figure 19.14 Effect menu.

Table 19.5 File Menu

Menu Item	Description
New	Creates a new empty project window.
Open	Presents you with a dialog box to choose a file to open. If a project window is open and empty, the new file will appear in that window; otherwise a new project window will open.
Close	Closes the current project window.
Save Project	Saves the current Audacity project file in AUP format. Audacity projects are not usable by other programs. Audio data for an Audacity project is not stored in the AUP file—instead, it is stored in a directory with the same name as the project.
Save Project As	Saves the current Audacity project file, with a different name or directory path.
Export as WAV	Exports the current Audacity project as a standard audio file format, such as WAV or AIFF. Change the format of the exported file in the Preferences dialog box.
Export Selection as WAV	The same as Export as WAV, but only exports the selected portion of the project.
Export as MP3	Exports the current Audacity project as an MP3 file. Exporting MP3 files requires that you install a separate MP3 encoder, not included with Audacity.
Export selection as MP3	The same as Export as MP3, but only exports the selected portion of the project.
Export as OGG	Exports the current Audacity project as an Ogg Vorbis file.
Export selection as OGG	The same as Export as OGG, but only exports the selected portion of the project.
Export Labels	Exports label tracks to a text file.
Preferences	Place where you configure Audacity.
Exit	Closes all project windows and exits Audacity. It will ask if you want to save changes to your project.

Team LRN

Table 19.6 Edit Menu

Menu Item	Description
Undo	Undoes the last edits performed. Allows you to undo every operation going back to the last time the document was saved.
Redo	Redoes edits that were just undone. The redo history remains available until you do a fresh edit.
Cut	Removes the selected audio data and moves it to the Clipboard.
Copy	Copies the selected audio data to the Clipboard.
Paste	Inserts the Clipboard contents at the position of the selection cursor in the project, replacing any selected data.
Delete	Removes selected data without copying it to the Clipboard.
Silence	Replaces selected audio data with silence.
Insert Silence	Inserts the prompted amount of silence entered at the current track cursor position in the project. Overwrites selections.
Split	Moves the selected region into its own track or tracks, replacing the affected portion of the original track with silence.
Duplicate	Makes a copy of all or part of a track or set of tracks into new tracks.
Select All	Selects all of the audio in all of the tracks.

Table 19.7 View Menu

Menu Item	Description
Zoom In	Zooms in on a portion of the audio data. Doing this allows you to view more data detail for a smaller time period.
Zoom Normal	Changes the zoom factor to one inch of data for one second of time; this is the default zoom factor.
Zoom Out	Zooms out so you can see a larger time base, at the cost of less detail.
Fit in Window	Adjusts the zoom factor so that the entire project fits exactly in the window.
Plot Spectrum	Plots a spectrum for a selected portion of a single track. First select a portion of audio data from a single track, then select this menu item. A window will open that will display the Power Spectrum of the audio for the selected data. The Power Spectrum computation uses the *Fast Fourier Transform* (FFT) algorithm to graph the proportional energy level for each frequency.
Float or Unfloat Palette	Allows you to switch between docking the Tool palette at the top of each project window or in a separate floating window.

Team LRN

Table 19.8 Project Menu

Menu Item	Description
Import Audio	Imports audio into your project. Use this function to add another track to a project with at least one existing track. You can also mix the imported track with an existing track.
Import Labels	Imports a text file that contains time codes and labels, turning them collectively into a label track.
Import MIDI	Imports a MIDI file into a note track. MIDI files can be viewed but not played, edited, or saved.
Import Raw Data	Allows you to open a file in virtually any uncompressed format. Audacity will examine the file contents to determine their format. You will need to listen to the result in order to decide if the program made the right format choice. You can use the dialog box the function displays to direct the program in its attempts. Sometimes a successful operation has a bit of noise at the beginning, caused by the unrecognized header format. The rest of the data usually plays correctly. You can then edit out the noise.
Edit ID3 Tags	Opens a dialog box allowing you to edit the ID3 tags associated with a project. These are used for MP3 exporting.
Quick Mix	Mixes selected tracks down to one or two tracks. Note that if you try to mix two loud tracks together, you may get clipping that sounds like pops, clicks, and noise. To avoid this, you should first adjust the gain (amplification) of the tracks to a lower level.
Align Tracks Together	Adjusts the time offset of selected multiple tracks to make them start at the same time. The starting time is adjusted to the average of all tracks' original starting times.
Align with Zero	Resets the time offset of elected multiple tracks to zero.
New Audio Track	Creates a new audio track with no data.
New Label Track	Creates a new label track.
Remove Track(s)	Removes the selected track or tracks from the project. You only need to select a portion of a track for it to be removed.
Add Label At Selection	Creates a new label at the current selection.

Team LRN

Table 19.9 Effect Menu

Menu Item	Description
Amplify	Increases or decreases the volume of a track or set of tracks. Audacity computes for you the maximum amount you could amplify the selected audio without being so loud that the signal is clipped.
BassBoost	Amplifies the lower frequencies yet leaves most of the other frequencies untouched. Recommended maximum boost is 12 dB.
Echo	Repeats the audio you have selected again and again, softer each time. There is a fixed time delay between each repeat.
Fade In	A linear fade-in to the selected audio data.
Fade Out	A linear fade-out to the selected audio.
FFT Filter	Applies a Fast Fourier Transform. The FFT Filter dialog is not properly finished, but the function works properly.
Invert	Flips the audio samples upside down.
Noise Removal	Removes constant background noise, such as fans, tape noise, or hums. Does not work well for removing background speech or music.
Phaser	Combines phase-shifted signals with the original signal.
Reverse	Reverses the selected audio temporarily; after the effect the end of the audio will be heard first and the beginning last. Useful for listening to old Beatles music!
Wahwah	Give us a little wah-wah, George! Uses a moving bandpass filter to create the famous wah-wah sound. This function also adjusts the phase of the left and right channels of a stereo recording, to make the effect travel between the speakers.
Freeverb2	Provides sliders to adjust reverb effects. Experiment with the sliders to achieve some interesting reverberating sounds.

Many Menu items can be invoked by the use of the standard Windows accelerator key combinations, such as Ctrl+W to close a window. Table 19.10 lists the shortcut keys.

note

The version of Audacity used here is version 1.0, which is a stable release but does lack a few features, the most notable of which is the ability to resample audio files.

You may be interested in version 1.2, which was released after this chapter was written. It includes many of the features missing from version 1.0. You can download Audacity 1.2 at http://audacity.sourceforge.net/.

Table 19.10 Shortcut Keys

Menu Item	Shortcut
File, New	Ctrl+N
File, Open	Ctrl+O
File, Close	Ctrl+W
File, Save Project	Ctrl+S
File, Preferences	Ctrl+P
Edit, Undo	Ctrl+Z
Edit, Redo	Ctrl+R
Edit, Cut	Ctrl+X
Edit, Copy	Ctrl+C
Edit, Paste	Ctrl+V
Edit, Delete	Ctrl+K
Edit, Silence	Ctrl+L
Edit, Split	Ctrl+Y
Edit, Duplicate	Ctrl+D
Edit, Select All	Ctrl+A
View, Zoom In	Ctrl+1
View, Zoom Normal	Ctrl+2
View, Zoom Out	Ctrl+3
View, Fit in Window	Ctrl+F
View, Plot Spectrum	Ctrl+U
Project, Import Audio	Ctrl+I
Project, Add Label at Selection	Ctrl+B

OpenAL

Torque, along with many other game engines, uses OpenAL—an open-source audio API (*Application Programmer's Interface*). In this book we won't be directly addressing programming with OpenAL, but we do need to ensure that OpenAL is installed with the correct version. The Torque installation procedure you followed in an earlier chapter will have taken care of that for you. In your root main directory, make sure that the file OpenAL32.dll is there. If it isn't, you will need to reinstall Torque. If you use the stock Options menu in the Torque demo games, then make sure you enable the OpenAL interface there.

Audio Profiles and Datablocks

Torque uses the concept of *datablocks* and *profiles* to help define and organize resources for use in the game. We encountered this concept when building our Emaga sample games in earlier chapters.

Team LRN

There are essentially two ways to make sounds occur in a Torque game. We can *directly* activate a sound (or music, for that matter) with program code, or we can attach sounds to in-game objects and let the Torque Engine activate and control the sounds *indirectly* on our behalf.

Most of the time we will use the latter—indirect—approach because once the relationship of sound-effects file to object has been defined in the right place, we don't need to worry about it anymore.

However, the first approach—direct activation—is more flexible. We'll look at both approaches in the remainder of this chapter.

Audio Descriptions

Audio datablocks are used no matter whether we directly or indirectly activate sounds. Audio datablocks are defined using the keyword `AudioDescription` when they are defined. Here is an example of an audio datablock:

```
new AudioDescription(AudioTest)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = false;
   type     = 0;
};
```

In this example, `AudioTest` is the handle to this description.

The `volume` property indicates the default volume for this channel. This property is itself not changeable, but when the audio channel is used, the volume can be changed via script statements.

The property `isLooping` indicates whether to repeat the sound after it has finished playing.

The `is3D` property is used to tell Torque whether this channel needs to be processed to produce positional information.

The `type` property is essentially the channel for this sound. All sounds on a given channel can be controlled via script statements that are channel specific.

With this datablock we have defined the nature of the `AudioTest` sound, so to speak—its characteristics. However, there's obviously not enough here to actually produce any sound. We need at least a sound file with a sample waveform in it, and then we need to associate that file with the appropriate `AudioDescription`. This is how we do it programmatically:

```
$Test = alxCreateSource("AudioTest",expandFilename("~/data/sound/test.wav"));
```

This statement creates an audio object. The first parameter is the datablock we saw earlier. The second parameter first invokes a call to the `expandFilename` function, which knows how to make sure it finds the correct full path of the file. The return value is a handle to the actual audio object created by Torque.

Now to activate the sound, we simply call the following:

```
alxPlay ($Test);
```

As you see, we just needed to tell `alxPlay` the name of the object, and away it goes.

We can adjust the volume for this playback, but we need to do it before we play the sound. We do that this way:

```
alxListenerf(AL_GAIN_LINEAR, %volume);
$Test = alxCreateSource("AudioTest",expandFilename("~/data/sound/test.wav"));
alxPlay ($Test);
```

The `alxListenerf` function sets the volume for the listener (the player) and does it using a linear (versus logarithmic) gain (amplification) adjustment. With a linear gain, a volume of 0.5 is half as loud as a volume of 1.0. With the nonlinear (logarithmic) gain, a volume of 0.5 is about two-thirds as loud as a volume of 1.0.

Note that this volume adjustment is performed on the value of the volume in the datablock, where the volume was set to 1.0.

So if we call `alxListenerf` with a volume of 0.75, then the actual volume would be 0.75 multiplied by 1.0, or 0.75—and all loudness calculations would follow from that. If we call `alxListenerf` with a volume of 0.75, and if the datablock's volume had been set to 0.5, then the actual volume would be 0.75 multiplied by 0.5, or 0.375.

Now using `alxPlay` this way is useful for sounds that have no positional information requirements, like GUI button beeps or the sound of a player's throbbing headache. But what if we want to place the sound in the game world?

In this case, we need to first create a profile:

```
new AudioProfile(AudioTestProfile)
{
  filename = "~/data/sound/test.wav";
  description = "AudioTest";
};
```

Notice that now the file name is contained in the profile. The second property, `description`, points to the datablock we defined earlier. We then activate the sound as follows:

```
alxPlay(AudioTestProfile, 100, 100, 100);
```

Notice now that the function call refers to the profile, not the description datablock. The three parameters that follow define a location in 3D coordinates in the game world. The sound, when played, will seem to come from that location. It's important to understand that when activating sounds in this manner, you must ensure that the sound file contains a monophonic sound, and not stereo. Also, the `is3D` property in the datablock must be set to false.

tip

Take note of whether you are creating the `AudioDescription` or `AudioProfile` on the client or the server.

On the client, you define it this way:

```
new AudioDescription(AudioTest)
{
};
```

and

```
new AudioProfile(AudioTestProfile)
{
}
```

If the code resides on the server, do it this way:

```
datablock AudioDescription(AudioTest)
{
};
```

and

```
datablock AudioProfile(AudioTestProfile)
{
}
```

In point of fact, this rule applies for all datablock types, because the server can only define true datablocks.

Trying It Out

Let's try it out, using your Emaga6 sample game. Open up your root main file (main.cs) and add the following lines to the very top:

```
new AudioDescription(AudioTest)
{
  volume   = 1.0;
  isLooping= false;
```

```
  is3D     = false;
  type     = 0;
};
new AudioProfile(AudioTestProfile)
{
  filename = "~/data/sound/test.wav";
  description = "AudioTest";
        preload = true;
};

function AudioTestA(%volume)
{
  echo("AudioTest volume="@%volume);
  alxListenerf(AL_GAIN_LINEAR, %volume);
  $pref::Audio::masterVolume = %volume;

    $AudioTestHandleA = alxCreateSource("AudioTest",
expandFilename("~/data/sound/test.wav"));

  echo("AudioTest object="@$AudioTestHandleA);
    alxPlay($AudioTestHandleA);

}

function AudioTestB(%volume)
{
  echo("AudioTest volume="@%volume);
  alxListenerf(AL_GAIN_LINEAR, %volume);
  $pref::Audio::masterVolume = %volume;
  alxPlay(AudioTestProfile, 100, 100, 100);
}
```

Now launch your game. After you've spawned in, open the console window (using the Tilde key) and type in the following:

```
AudioTestA(1.0);
```

You should hear the "electronic drip" test sound. Play with the volume setting, trying different values less than 1.0.

Next type this into the console window:

```
AudioTestB(1.0);
```

You should hear the electronic drip test sound again, but this time seeming to come from a specific direction.

Again, play with the volume setting, trying different values less than 1.0. You can also play with the 3D coordinate values in the call to `alxPlay()` in the `AudioTestB()` function.

Koob

In the following chapter, and in later chapters, we will be using audio features a great deal more, so take the time in the balance of this chapter to add some more files to your sample program.

First, copy your Emaga6 directory and name the copy "koob". Or use any other name—but I'll be using koob.

Now create a new directory: C:\koob\control\data\sound. Record a sound, any sound, in a WAV file. Make sure that the file is not a stereo file. Copy your new sound file into C:\koob\control\data\sound and name it "test.wav".

Next create a new script file: C:\koob\control\client\misc\sndprofiles.cs. Insert the following lines of code:

```
// Channel assignments (channel 0 is unused in-game).
$GuiAudioType     = 1;
$SimAudioType     = 2;
$MessageAudioType = 3;

new AudioDescription(AudioGui)
{
  volume   = 1.0;
  isLooping= false;
  is3D     = false;
  type     = $GuiAudioType;
};

new AudioDescription(AudioMessage)
{
  volume   = 1.0;
  isLooping= false;
  is3D     = false;
  type     = $MessageAudioType;
};

new AudioProfile(AudioButtonOver)
{
```

```
  filename = "~/data/sound/buttonOver.wav";
  description = "AudioGui";
  preload = true;
};
```

This sets up some datablocks and a profile for use on our client.

Next create a new script file: C:\koob\control\server\misc\sndprofiles.cs. Insert the following lines of code:

```
datablock AudioDescription(AudioDefault3d)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = true;
   ReferenceDistance= 20.0;
   MaxDistance= 100.0;
   type     = $SimAudioType;
};

datablock AudioDescription(AudioClose3d)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = true;
   ReferenceDistance= 10.0;
   MaxDistance= 60.0;
   type     = $SimAudioType;
};

datablock AudioDescription(AudioClosest3d)
{
   volume   = 1.0;
   isLooping= false;
   is3D     = true;
   ReferenceDistance= 5.0;
   MaxDistance= 30.0;
   type     = $SimAudioType;
};

// Looping sounds
datablock AudioDescription(AudioDefaultLooping3d)
{
   volume   = 1.0;
```

```
   isLooping= true;
   is3D     = true;
   ReferenceDistance= 20.0;
   MaxDistance= 100.0;
   type     = $SimAudioType;
};

datablock AudioDescription(AudioCloseLooping3d)
{
   volume   = 1.0;
   isLooping= true;
   is3D     = true;
   ReferenceDistance= 10.0;
   MaxDistance= 50.0;
   type     = $SimAudioType;
};

datablock AudioDescription(AudioClosestLooping3d)
{
   volume   = 1.0;
   isLooping= true;
   is3D     = true;
   ReferenceDistance= 5.0;
   MaxDistance= 30.0;
   type     = $SimAudioType;
};

// Used for non-looping environmental sounds (like power on, power off)
datablock AudioDescription(Audio2D)
{
   volume = 1.0;
   isLooping = false;
   is3D = false;
   type = $SimAudioType;
};

datablock AudioDescription(AudioLooping2D)
{
```

```
   volume = 1.0;
   isLooping = true;
   is3D = false;
   type = $SimAudioType;
};
```

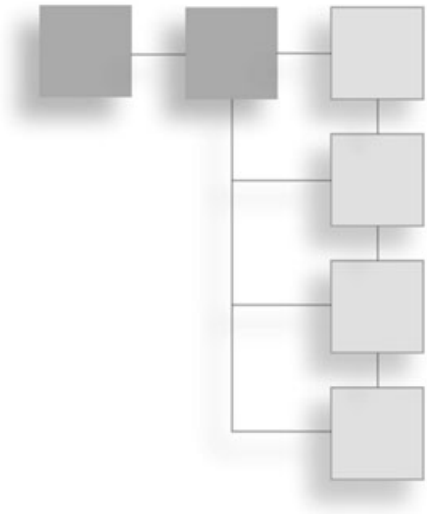
All of this sets up some datablocks for the server—we will use them in the next chapter. I include them here for you to peruse within the context of what we've covered in this chapter. For practice, you can try various calls to `alxPlay` and create some profiles that use these descriptions.

Moving Right Along

In this chapter, you've explored a new tool—this time for dealing with sound. You've learned how to create and export a WAV file for use in a game, and then how to insert a sound into a Torque game. You've also seen how you can adjust a sound using Torque Script, as well as position a sound in the 3D game world.

In the next chapter we'll look at sound effect in the game world in more detail.

CHAPTER 20



Game Sound and Music

In the last chapter you saw how to create and edit sounds using Audacity. We're now going to move to the next level, using those sounds for effects initiated by the player, by weapons and ammo, by vehicles, and by places and things in the world at large.

Also, we'll touch on the issue of in-game music and how you can use it. I'm not going to even attempt to teach you how to compose music—that's far out of scope for this book. However, I will include some musical pieces that you are free to experiment with as we look at the issue.

Player Sounds

In a first-person shooter-styled game, player sounds add to the sense of "being there," sometimes in a big way. There are two kinds of player sounds: world sounds and client-only sounds.

World sounds are effects that are generated on the server that represent sound effects emitted by your player-avatar in the game world. In this sense, they are much the analogue of the way you emit sounds in real life: walking, talking, firing weapons, banging on doors, and so on. The server places a sound effect "in the world" at your location and then updates all of the affected clients so that they will emit the sound (if the client's player-avatars are close enough to hear the sound) as it was made, with appropriate modifications, such as attenuation because of distance or Doppler effect due to the sound source moving toward or away from the listener.

These sorts of sounds are usually called *3D sounds*. The actual sound effects have no inherent 3D characteristics, but the game client handles them in a manner that imparts 3D positional information to each client.

Team LRN

Client-only sounds are those sounds that a player's avatar makes that really only matter to the player. These can be personal noises, like the sounds of heavy breathing to portray exertion, the sound of being hit by a bullet, or the rustle of clothing. Nothing is cut-and-dried though—you might want to use the sounds in some of those examples as world sounds in order to perhaps betray the location of a player who is sneaking around in the dark. It all depends on your game play design.

Some sounds can be attached to the player, like the sounds of footsteps, triggered by a frame in the animation. Unfortunately for us in this book, the MilkShape DTS Exporter does not yet support this capability. I happen to know that this problem will be overcome in the near future, but alas, the capability to trigger footsteps using frames in model animations is not yet available. Triggered footsteps *are* available if you use the more expensive modeling tools that cost in the range from several hundred to several thousand dollars. The DTS exporters for those tools *do* support animation triggers. But like I said, the capability will be available soon for the MilkShape DTS Exporter.

However, that being said, there is a way for us to handle footsteps using program code, and that's what we'll be doing a little bit later in the chapter.

Other sounds can be emitted in an ad hoc fashion, wherever and whenever we want. Some examples of these sorts of sounds are utterances or taunts. You press a key and your player's avatar utters a taunt of some sort, such as "Loser!" or "Ha ha, ya missed me!" You can also use ad hoc sounds to issue prerecorded audible commands. The limits are your imagination. Heck, you could have your player carry around a boom box and play annoying music!

When you use world sounds, you also have the choice of using them in a 3D mode, where the sound is played on all the clients as if they were emanating from a specific location in the game world, or you can use them in 2D mode—the server still directs the clients to play the sounds, but they have no positional quality to them.

Footsteps

In our first example, we are going to use the `serverPlay3D` function to make out footsteps for our player. First you should record some footsteps. Just a single one will be enough, if you like, but if you record a half dozen or so that are all similar though slightly different, you can offer a more natural sound effect by randomly choosing which sound to play for a given footfall. Make sure to record the sound at 22050 Hertz (Hz) or maybe even 11025 Hz to keep the file size fairly small. Save the sound as C:\koob\control\data\sound\footstep1.wav.

After you have the sound effect made, you need to add the following code in C:\koob\control\server\players\player.cs at the end of the file (after the function `HumanMaleAvatar::onDisabled`).

```
datablock AudioProfile(FootStep1)
{
   fileName = "~/data/sound/FootStep1.wav";
   description = AudioClosest3d;
   preload = true;
};

function serverCmdStartFootsteps(%client)
{
  %client.player.schedule(200,playFootstep);
  %client.player.footstepson = true;
}
function serverCmdStopFootsteps(%client)
{
  %client.player.footstepson = false;
}

function Player::playFootstep( %this )
{
  if(%this.footstepson)
  {
    serverPlay3D(FootStep1,%this.getTransform());
    %this.schedule(500,playFootstep);
  }
}
```

First, there is an `AudioProfile` datablock. This datablock tells the engine where the sound effect is and which `AudioDescription` to use. The particular `AudioDescription` in question already resides in C:\koob\control\server\misc\sndprofiles.cs and looks like the following (do not type this in):

```
datablock AudioDescription(AudioClosest3d)
{
   volume   = 1.0;
   isLooping= false;

   is3D     = true;
   ReferenceDistance= 5.0;
   MaxDistance= 30.0;
   type     = $SimAudioType;
};
```

The next thing in the new code we added was a message handler for receiving a message from a client. We defined the message in this case to be `StartFootsteps`, and our only

parameter is the handle to the client that sends the message. That handler makes a call to a method of the `Player` object called `schedule`. This schedules a function execution event for processing sometime later. We also set the flag `%client.player.footstepson` to `true` for future reference. The event delay is set to 500 milliseconds, or half a second, in the future. You can change this value to something else or even have it vary according to running speed. At the appointed time, that function, `playFootstep`, is called, and provided that the `%this.footstepson` property is set to `true`, it executes the `serverPlay3D` function. Note that in this case, `%this` is set to be the handle of the object the method is being called for. That object is the `Player` object, which happens to be exactly the same object as `%client.player`, and it's a good thing that it is too!

The way `serverPlay3D` works is that it accepts an `AudioProfile` and 3D coordinates in the world space. Conveniently we can get those coordinates from a simple call to `getTransform`. The `serverPlay3D` function then internally tells all clients to play that sound effect at those world coordinates.

And hey, presto! You have a footstep.

Before exiting `playFootstep`, the `schedule` method is called again to schedule another footstep in half a second. It will keep doing this until told to stop. You tell it to stop by using the `stopFootsteps` message, whose handler merely sets `%client.player.footstepson` to `false`, so that the next time `playFootstep` is called, the flag is found to be false, the sound is played, and the event isn't rescheduled.

So that's the guts of getting the sounds to play, but we still need to deal with *when* to play them. We want the steps to happen when the player is running and to stop when he stops.

We can easily do this as part of the work-around by trapping the keyboard inputs that tell the player to move and stop. The function that does this is a client-side function located in C:\koob\control\client\misc\presetkeys.cs.

Open that file and locate the function `GoAhead`, which looks like this:

```
function GoAhead(%val)
//------------------------------------------------------------------------
// running forward
//------------------------------------------------------------------------
{
   $mvForwardAction = %val;
}
```

Change it to this:

```
function GoAhead(%val)
//----------------------------------------------------------------------------
// running forward
//----------------------------------------------------------------------------
{
   $mvForwardAction = %val;
   if (%val)
      commandToServer('startFootsteps');
   else
      commandToServer('stopFootsteps');
}
```

In `GoAhead`, the parameter `%val` is nonzero when the key that has been mapped to this function is being pressed, and it is zero when the key is released. Therefore, the simple `if-else` code block will send the `startFootsteps` message to the server when the GoAhead key is pressed and the `stopFootsteps` message when it is released. The GoAhead key is defined later in the same file to be the Up Arrow key.

Now if you have been browsing around sndProfiles.cs looking at the datablocks in there, you might have come across another work-around you're tempted to try—the `AudioClosestLooping3d` datablock. You might look at that and say to yourself, "Self, that has looping *built in*. No need to fool with scheduling events on a repeating basis." And you would be right in making that deduction. However, there is a problem with that approach: Once you trigger that sound effect at a particular location, it will continue looping, all right—but *at the same location*. The footsteps won't follow your player.

Like I said earlier, the absolute best way to do these kinds of repetitive player sounds is to attach them to the player's movement animations via triggers in the model. For MilkShape users, that capability will be available soon.

Utterances

Let's make our avatar guy say something, something that other players can hear, by pressing a key. The process is going to be quite similar to the footsteps.

First, make another recording, at the sample rate of your choosing. Holler something into the mike, like, "Your mother wears army boots!" or something equally endearing. Save it as C:\koob\control\data\sound\insult1.wav or something like that.

Then add the following code in C:\koob\control\server\players\player.cs at the end of the file.

```
datablock AudioProfile(Insult1)
{
   fileName = "~/data/sound/insult1.wav";
   description = AudioClose3d;
```

```
   preload = true;
};

function serverCmdHurlInsult(%client)
{
  serverPlay3D(Insult1,%this.getTransform());
}
```

There is noticeably less stuff than for the footsteps, because we don't need to make the sound effect. We just hurl our insult and maybe run for cover after that! Now, notice that the profile uses a different `AudioDescription`. This time it's `AudioClose3d`. (Don't type this in—it's already in sndProfiles.cs.)

```
datablock AudioDescription(AudioClose3d)
{
   volume   = 1.0;
   isLooping= false;

   is3D     = true;
   ReferenceDistance= 10.0;
   MaxDistance= 60.0;
   type     = $SimAudioType;
};
```

The reason for using this datablock is because it defines a sound effect that can be heard from farther away. The `ReferenceDistance` is 10 world units. This means that the sound effect *attenuates* (the volume decreases) over a longer distance, so it can be heard from farther away than the footsteps.

Next, we need to send the message from the client to the server so that the server can then notify all the other clients. We'll do that again with a client-side function that we'll call `Yell`.

Open C:\koob\control\client\misc\presetkeys.cs and add the following to the end:

```
function Yell(%val)
{
   if (%val)
commandToServer('HurlInsult');
}
PlayerKeymap.bind(keyboard, "y", Yell);
```

The function sends the `HurlInsult` message to the server, but only when the key is pressed (`%val` is nonzero), not when it's released. Then we need to bind a key to press to trigger the whole thing. We use `PlayerKeymap.bind` to do that, pointing it to the `Yell` function.

There you go—you're in business.

Team LRN

One more variation you should try is recording several different insults and saving them as insult1.wav, insult2.wav, and so on. Let's go ahead and record six different insults.

Now make six different `AudioProfiles` that have incremental names starting with `Insult1` and ending with `Insult6`. Each should uniquely point at one of the six recordings you made. Then in the message handler use a bit of random number code to pick a number between 1 and 6.

```
%n=getRandom(5) + 1;
```

This would pick an integer between 0 and 5. Now increment it by 1 so that the result would be between 1 and 6.

Then rewrite `serverPlay3D` to look like this:

```
serverPlay3D("Insult" @ %n, %this.getTransform());
```

This will modify the name of the `AudioProfile` by putting the random number at the end. Then every time you hurl the insult, a different epithet will be directed with withering precision on your foe. Fun for the whole family!

Weapon Sounds

Weapon sounds are an interesting study. Weapons have specific support in Torque, through the use of a programming construct called a *state machine*. The basic idea is that we break the operation of a weapon down into different stages, called *states*, and we define a specific set of behaviors for each state. Within each state, we are not aware of what the previous state was, only what needs to be done in this state.

Using this system, we can quite readily define some rather complex behaviors.

To set up for this, go find the Tommy gun model you created back in Chapter 16, and copy the model (the DTS file) and the artwork (the PNG file) that goes with the model to C:\koob\control\data\models\weapons\. Then locate the directory C:\3DGPAi1\RESOURCES\CH20\ and copy the file tommygun.cs into C:\koob\control\server\weapons\.

Next, from the same directory, copy the following files to C:\koob\control\data\models\weapons\:

```
ammo.jpg
bullethole.png
muzzleflash.png
tgammo.dts
tgprojectile.dts
tgshell.dts
```

```
tommygun.cs
tommygun.dts
tommygun.jpg
```

Now for the sounds. I'm not going to make you record your own sounds. You can copy them from the same directory.

```
ammo_pickup.wav
dry_fire.wav
short_reload.wav
tommy_gun.wav
weapon_pickup.wav
weapon_switch.wav
```

Deposit these sound files into C:\koob\control\data\sound\.

Next, open the file C:\koob\control\server\server.cs and find the line that reads as follows:

```
$Game::StartTime = 0;
```

Just beyond that line is a block of exec() statements. Insert the following at the top or bottom of that block of statements:

```
Exec("./weapons/tommygun.cs");
```

This tells the engine to load our Tommy gun definition file.

Next, open the file C:\koob\control\server\players\player.cs and find the line that reads as follows:

```
datablock PlayerData(HumanMaleAvatar)
```

At the end of the datablock that starts with that line, before the closing brace ("}") that ends the datablock, insert the following lines:

```
   maxInv[Tommygun] = 1;
   maxInv[TommygunAmmo] = 20;
```

This indicates how many of the listed items the player can have in his possession, or inventory, at any given time.

And finally, open the file you copied earlier, C:\koob\control\server\weapons\tommygun.cs, and add the following lines to the end:

```
datablock AudioProfile(TommyGunMountSound)
{
   filename    = "~/data/sound/shortreload.wav";
   description = AudioClose3d;
   preload = true;
```

```
};

datablock AudioProfile(TommyGunReloadSound)
{
   filename    = "~/data/sound/shortreload.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(TommyGunFireSound)
{
   filename    = "~/data/sound/tommygun.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(TommyGunDryFireSound)
{
   filename    = "~/data/sound/dryfire.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(WeaponSwitchSound)
{
   filename    = "~/data/sound/Weapon_switch.wav";
   description = AudioClose3d;
   preload = true;
};


//--------------------------------------------------------------------------
// TommyGun image which does all the work.  Images do not normally exist in
// the world, they can only be mounted on ShapeBase objects.

datablock ShapeBaseImageData(TommyGunImage)
{
   shapeFile = "~/data/models/weapons/TommyGun.dts";
   offset = "0 0 0";
   mountPoint = 0;
   emap = true;
```

```
className = "WeaponImage";

item = TommyGun;
ammo = TommyGunAmmo;
projectile = TommyGunProjectile;
projectileType = Projectile;
casing = TommyGunShell;
armThread = "look2";

// State Data
stateName[0]                        = "Preactivate";
stateTransitionOnLoaded[0]       = "Activate";
stateTransitionOnNoAmmo[0]       = "NoAmmo";

stateName[1]                        = "Activate";
stateTransitionOnTimeout[1]      = "Ready";
stateTimeoutValue[1]             = 0.7;
stateSequence[1]                 = "Activated";
stateSound[1]                       = WeaponSwitchSound;

stateName[2]                        = "Ready";
stateTransitionOnNoAmmo[2]       = "NoAmmo";
stateTransitionOnTriggerDown[2]  = "Fire";
stateScript[2]                      = "onReady";
stateTransitionOnReload[2]       = "Reload";


stateName[3]                        = "Fire";
stateTransitionOnTimeout[3]      = "Ready";
stateTimeoutValue[3]             = 0.096;
stateFire[3]                        = true;
stateRecoil[3]                      = LightRecoil;
stateAllowImageChange[3]         = false;
stateSequence[3]                 = "Fire";
stateScript[3]                      = "onFire";
stateSound[3]                       = TommyGunFireSound;
stateEmitter[3]                  = TommyGunFireEmitter;
stateEmitterTime[3]              = 1.0;
stateEmitterNode[3]              = "muzzlePoint";

stateName[4]                        = "Reload";
stateTransitionOnNoAmmo[4]       = "NoAmmo";
```

```
   stateTransitionOnTimeout[4]      = "FinishedReloading";
   stateTimeoutValue[4]              = 3.5; // 0.25 load, 0.25 spinup
   stateAllowImageChange[4]         = false;
   stateSequence[4]                  = "Reload";
   stateScript[4]                    = "onReload";
   stateSound[4]                     = TommyGunReloadSound;


   stateName[5]                      = "FinishedReloading";
   stateTransitionOnTimeout[5]      = "Activate";
   stateTimeoutValue[5]              = 0.04;
   stateScript[5]                    = "onFinishedReloading";

   stateName[6]                      = "NoAmmo";
   stateTransitionOnAmmo[6]         = "Reload";
   stateSequence[6]                  = "NoAmmo";
   stateScript[6]                    = "onNoAmmo";
   stateTransitionOnTriggerDown[6]  = "DryFire";

   stateName[7]                      = "DryFire";
   stateSound[7]                     = TommyGunDryFireSound;
   stateScript[7]                    = "onDryFire";
   stateTimeoutValue[7]              = 0.5;
   stateTransitionOnTimeout[7]      = "NoAmmo";

   stateName[8]                      = "WaitTriggerRelease";
   stateScript[8]                    = "onWaitTriggerRelease";
   stateTransitionOnTimeout[8]      = "WaitTriggerRelease";
   stateTimeoutValue[8]              = 0.01;
   stateTransitionOnTriggerUp[8]    = "Ready";

   autoFire = true;
   weaponDamage = 60;
   minSpread = 0.01;
   maxSpread = 0.045;
   spreadRate = 0.019;   // amount spread should increase per shot
   spreadRecoverRate = 0.003;
};
```

The first thing this new code does is define a bunch of audio profiles, `TommyGunMountSound`, `TommyGunReloadSound`, `TommyGunFireSound`, `TommyGunDryFireSound`, and `WeaponSwitchSound`. These profiles are used in each of the different weapon firing states. Those states are defined in the next part of the new code.

Team LRN

That next part is a datablock of the type `ShapeBaseImageData`. This is what defines the gun itself and how it works.

First, there is a set of basic properties, like where to find the model that represents the image and so on. For this example, I have used the same model as the one that is used for the external view—the view of your player model that everyone else sees. You, though, only see the weapon image. This means that to do this right, you will need to make another model of the weapon for use in this image. Later on you will see why this matters.

Now we add the `WeaponImage` name space as a parent. The `WeaponImage` name space provides some hooks into the inventory system that are necessary for picking up the gun.

Next are a bunch of pointers that tell what various resources we will need in order to use this gun.

Finally, we encounter the code that defines the state machine. What happens is that when you pick up the gun, the Torque Engine sets it to the first state: `Preactivate`.

In the `Preactivate` state, we have only two variables, and they tell the state machine what to do immediately next. If the gun is loaded, it should change to the `Activate` state; if not, it should change to the `NoAmmo` state. If you scroll down until you find the line that says `stateName[6] = "NoAmmo";` you will find that state's definition.

In the `NoAmmo` state, there are several directives that the engine must follow while in this state. If we suddenly receive some ammo, then we change to the `Reload` state. If the gun's trigger is pressed, we enter the `DryFire` state. Note that there is also a pointer to a function (the `onNoAmmo` function) that we can execute when we find ourselves in this state. This can also be called the *state handler*.

All of the rest of the states operate in a similar way, and the directives are quite easy to read and follow. The important ones for this chapter are the `stateSound` directives, which tell the engine which audio profiles to use when we arrive in that state.

The state machine definition in the `TommyGunImage` datablock you've just seen is really quite easy to follow. You can modify it in all sorts of ways to accommodate any variation you can imagine.

Now after getting C:\koob\control\server\weapons\tommygun.cs typed in and double-checking it all, let's try it out.

Launch your Koob game. Once you have spawned in, we are going to use the World Editor to insert a Tommy gun and some ammo into the game world.

1. Press F8. This will set your player into camera fly mode.
2. Press F11. This will open up the World Editor, as shown in Figure 20.1.

3. Press F4. This will open up the World Editor Creator, as shown in Figure 20.2. The Creator frame is at the lower-right corner of the window.
4. In the Creator frame, click the plus sign next to the entry Shapes. This will expand the listing.
5. Locate Weapon and click the plus sign to open it as well. You should now have a Tree view similar to Figure 20.3.
6. Make sure that the center of the view is located in an open area about 10 virtual feet in front of you. To move the view in the World Editor, hold down the right mouse button, and move the mouse.
7. Click Tommygun in the Tree view. The Tommy gun model will appear; it will probably be somewhat embedded in the ground, as shown in Figure 20.4, and it will be rotating.
8. Move the cursor over on top of the vertical line (labeled Z) that sprouts from the top of the gun model. The Z label will become highlighted, as shown in Figure 20.5.
9. Click the vertical Z line and drag it up just a few pixels, until the gun is completely out of the ground, as depicted in Figure 20.6.

   Note that this is the reason why you needed to switch to camera fly mode before entering the World Editor. If you had stayed in normal FPS view mode, you would not have been able to grab the Z line and move it.

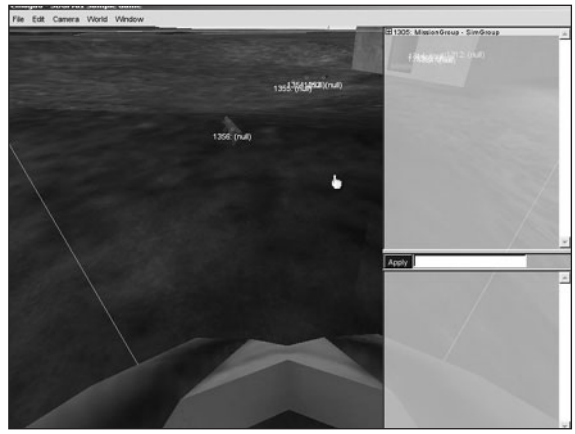


Figure 20.1 World Editor.

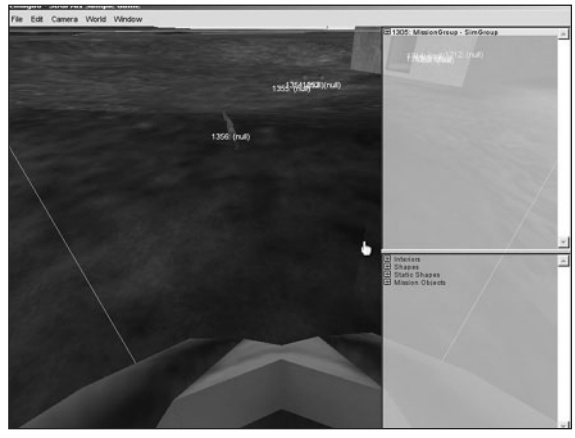


Figure 20.2 World Editor Creator.

Figure 20.3 The Creator Tree view.

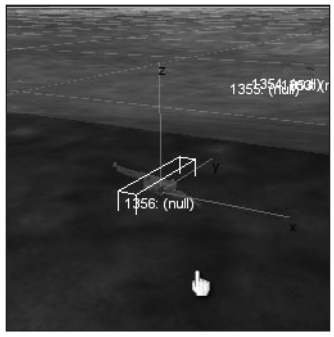


Figure 20.4 Tommy gun model.

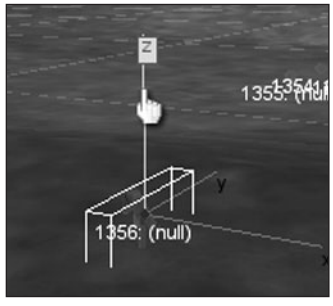


Figure 20.5 The Z label.

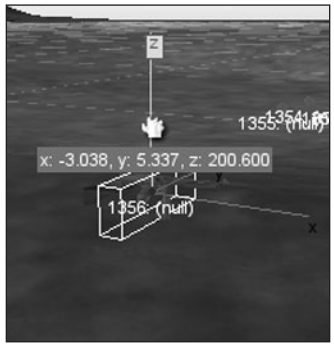


Figure 20.6 Repositioned Tommy gun.

10. Now turn your view slightly to the side, and repeat the same process by placing an ammo box, as shown in Figure 20.7. The ammo box can be found in the Tree view at Shapes, Ammo, TommygunAmmo.
11. Now press F11 to toggle out of the World Editor.

Okay, now run on over and pick up the ammo and the Tommy gun by just passing right over on top of them.

You will immediately notice that the gun doesn't carry properly. However, go ahead and shoot it, and listen to the firing sequence and all of the sounds we've been dealing with. You can make another model to act as the mounted (carried) version of the gun. Also, you will need to adjust your model animations to ensure the model will carry the gun properly—the provided example character doesn't do that.

You can go back to your `ShapeBaseImageData` in the tommygun.cs file and fiddle with the state machine and other variables and see how they affect your gun's behavior.

Vehicle Sounds

Vehicles are obvious sound sources. An idling engine, squealing tires, whirring propellers—the kind of vehicle dictates the needs. Torque has several defined vehicle types, but we'll just look at the wheeled vehicle and add some sound effects to the runabout.

To start, you will need to record sound effects for the following:

- engine idle
- acceleration
- wheel impact
- wheel squeal
- soft crash
- hard crash

In lieu of creating your own, feel free to use the sounds that I have provided at C:\3DGPAi1\RESOURCES\CH20\. Deposit the files into C:\koob\control\data\sound\.

Next, copy the car definition module, C:\3DGPAi1\RESOURCES\CH20\CAR.CS, to C:\koob\control\data\server\vehicles. If the directory doesn't exist, create it.

Then copy your runabout model and all its artwork into C:\koob\control\data\models\vehicles. Again, if the directory doesn't exist, create it. Make sure your runabout is named "runabout.dts" and the wheel model is named "wheel.dts".

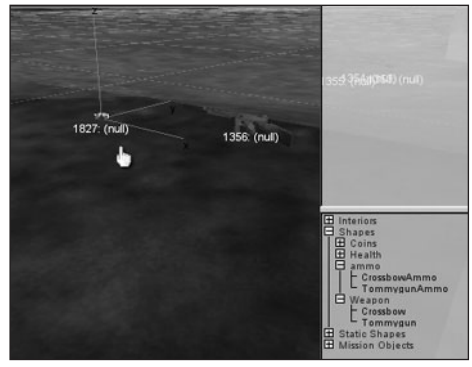


Figure 20.7 Placing ammo box.

Now open the file C:\koob\control\server\server.cs and find the line that reads as follows:

```
$Game::StartTime = 0;
```

Just beyond that line is a block of `exec()` statements. Insert the following at the top or bottom of that block of statements:

```
Exec("./vehicles/car.cs");
```

This tells the engine to load your car definition file.

And finally, open the file you copied earlier, C:\koob\control\server\vehicles\car.cs, and add the following lines to the end:

```
datablock AudioProfile(CarSoftImpactSound)
{
   filename    = "~/data/sound/vcrunch.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(CarHardImpactSound)
{
   filename    = "~/data/sound/vcrash.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(CarWheelImpactSound)
{
   filename    = "~/data/sound/impact.wav";
   description = AudioClose3d;
```

```
   preload = true;
};

datablock AudioProfile(CarThrustSound)
{
   filename    = "~/data/sound/caraccel.wav";
   description = AudioDefaultLooping3d;
   preload = true;
};

datablock AudioProfile(CarEngineSound)
{
   filename    = "~/data/sound/caridle.wav";
   description = AudioClose3d;
   preload = true;
};

datablock AudioProfile(CarSquealSound)
{
   filename    = "~/data/sound/squeal.wav";
   description = AudioClose3d;
   preload = true;
};

datablock WheeledVehicleData(DefaultCar)
{
   category = "Vehicles";
   className = "Car";
   shapeFile = "~/data/models/vehicles/runabout.dts";
   emap = true;

   maxDamage = 1.0;
   destroyedLevel = 0.5;

   maxSteeringAngle = 0.785;  // Maximum steering angle
   tireEmitter = TireEmitter; // All the tires use the same dust emitter

   // 3rd person camera settings
   cameraRoll = true;          // Roll the camera with the vehicle
   cameraMaxDist = 6;          // Far distance from vehicle
   cameraOffset = 1.5;         // Vertical offset from camera mount point
   cameraLag = 0.1;            // Velocity lag of camera
```

```
   cameraDecay = 0.75;         // Decay per sec. rate of velocity lag

   // Rigid Body
   mass = 200;
   massCenter = "0 -0.5 0";     // Center of mass for rigid body
   massBox = "0 0 0";           // Size of box used for moment of inertia,
                                 // if zero it defaults to object bounding box
   drag = 0.6;                  // Drag coefficient
   bodyFriction = 0.6;
   bodyRestitution = 0.4;
   minImpactSpeed = 5;         // Impacts over this invoke the script callback
   softImpactSpeed = 5;        // Play SoftImpact Sound
   hardImpactSpeed = 15;       // Play HardImpact Sound
   integration = 4;            // Physics integration: TickSec/Rate
   collisionTol = 0.1;         // Collision distance tolerance
   contactTol = 0.1;           // Contact velocity tolerance

   // Engine
   engineTorque = 4000;        // Engine power
   engineBrake = 600;          // Braking when throttle is 0
   brakeTorque = 8000;         // When brakes are applied
   maxWheelSpeed = 30;         // Engine scale by current speed / max speed

   // Energy
   maxEnergy = 100;
   jetForce = 3000;
   minJetEnergy = 30;
   jetEnergyDrain = 2;

   // Sounds
   engineSound = CarEngineSound;
   jetSound = CarThrustSound;
   squealSound = CarSquealSound;
   softImpactSound = CarSoftImpactSound;
   hardImpactSound = CarHardImpactSound;
   wheelImpactSound = CarWheelImpactSound;
};
```

As you've seen in earlier sections, we start out with a gaggle of `AudioProfiles` that define each of our sounds.

After that comes the vehicle datablock. Most of the properties are explained in the code commentary or are self-explanatory. The ones that we are interested in the most are at the end.

The `engineSound` property is the sound the vehicle makes while idling. As long as the vehicle is running, it will make this noise.

The `jetSound` property is the one used when the vehicle accelerates. The name is a holdover from the *Tribes 2* game engine in the early Torque days.

The `squealSound` property is the sound emitted by the tires when the vehicle is manhandled around a corner, causing the tires to slip.

The two impact sound properties, `softImpactSound` and `hardImpactSound`, are used when the vehicle collides with objects at different speeds, as defined by the `softImpactSpeed` and `hardImpactSpeed` properties earlier in the datablock.

Finally, the `wheelImpactSound` is the sound emitted when the wheels hit something at greater than the minimum impact speed, defined by `minImpactSpeed` earlier in the datablock.

Now we have to make some changes to our player's behavior. What we want is to have the player get in the car when he goes up to it.

Open the file C:\koob\control\server\players\player.cs and locate this line:

```
%this = %col.getDataBlock();
```

and add the following after it:

```
   if ( %this.className $= "Car" )
   {
       if (%this.salvageFlag)
                       %this.salvageFlag=0;
       %node = 0;   // Find next available seat
       %col.mountObject(%obj,%node);
       %obj.mVehicle = %col;
   }
   else
   {
```

Then scroll down until you find this line:

```
   %col.applyImpulse(%pos,%vec);
```

and add a closing brace ("}") after that line.

Next, add the following code to the end of the file:

```
function HumanMaleAvatar::onMount(%this,%obj,%vehicle,%node)
{
   %obj.setTransform("0 0 0 0 0 1 0");
   %obj.setActionThread(%vehicle.getDatablock().mountPose[%node]);
   if (%node == 0)
```

```
   {
      %obj.setControlObject(%vehicle);
      %obj.lastWeapon = %obj.getMountedImage($WeaponSlot);
      %obj.unmountImage($WeaponSlot);
      %db = %vehicle.getDatablock();
   }
}

function HumanMaleAvatar::onUnmount( %this, %obj, %vehicle, %node )
{
   %obj.mountImage(%obj.lastWeapon, $WeaponSlot);
}

function HumanMaleAvatar::doDismount(%this, %obj, %forced)
{
   // This function is called by the game engine when the jump trigger
   // is true while mounted

   // Position above dismount point
   %pos    = getWords(%obj.getTransform(), 0, 2);
   %oldPos = %pos;

   %vec[0] = " 1  1  1";
   %vec[1] = " 1  1  1";
   %vec[2] = " 1  1  -1";
   %vec[3] = " 1  0  0";
   %vec[4] = "-1  0  0";

   %impulseVec  = "0 0 0";
   %vec[0] = MatrixMulVector( %obj.getTransform(), %vec[0]);

   // Make sure the point is valid
   %pos = "0 0 0";
   %numAttempts = 5;
   %success     = -1;

   for (%i = 0; %i < %numAttempts; %i++)
   {
      %pos = VectorAdd(%oldPos, VectorScale(%vec[%i], 3));
      if (%obj.checkDismountPoint(%oldPos, %pos))
      {
         %success = %i;
```

```
            %impulseVec = %vec[%i];
            break;
         }
      }
      if (%forced && %success == -1)
         %pos = %oldPos;
      %obj.unmount();
      %obj.setControlObject(%obj);
      %obj.mountVehicle = false;

      // Position above dismount point
      %obj.setTransform(%pos);
      %obj.applyImpulse(%pos, VectorScale(%impulseVec, %obj.getDataBlock().mass));
}
```

This code allows us to get in (mount) the car and then get out (dismount). There's nothing about the sound in there, but it is convenient to have this ability.

Now use the same procedures with the World Editor as with the Tommy gun to insert the car into the game world. You will find the car in the Tree view under Shapes, Vehicles. Remember to tug the model up out of the ground if it's embedded in the ground—but don't tug it too hard!

Run up to the car, and you will automatically go inside and be seated. Use the normal forward movement key (Up Arrow) to accelerate, and the mouse to steer left and right. Have at it!

Environmental Sounds

A silent world is a dreary one. You can liven up your game world by inserting sounds to give a sense of the environment using AudioEmitters.

First, copy the file C:\3DGPAi1\RESOURCES\CH20\LOON.WAV over to C:\koob\control\data\sounds\.

Then open C:\koob\control\server\misc\sndprofiles.cs and add the following AudioProfile to the end of the file:

```
datablock AudioProfile(LoonSound)
{
   filename    = "~/data/sound/loon.wav";
   description = AudioDefaultLooping3d;
};
```

Run your game, and open the World Editor and then the World Editor Creator as before.

Next, in the Tree view, locate Mission Objects, environment, AudioEmitter as shown in Figure 20.8. Click AudioEmitter while facing a location where you would like to place the AudioEmitter.

In the dialog box presented (see Figure 20.9), click the Sound Profile button. From the list that opens, choose the LoonSound Profile.

Make sure the Use profile's desc?, Looping?, and Is 3D sound? check boxes are selected, and then click OK. Check Figure 20.9 to verify the settings.

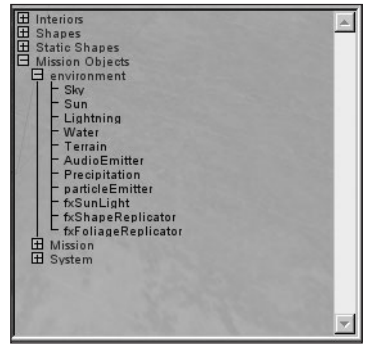


Figure 20.8 AudioEmitter in Tree view.

An AudioEmitter marker will be placed in the game world at the center of your screen, on the ground, as shown in Figure 20.10.

Now exit the editor by toggling the F11 key, make sure you are in camera fly mode, and move up and away from where you placed the marker. Then go back in to the editor. You should see two concentric spheres, as shown in Figure 20.11. The inner sphere is very faintly defined with gray lines in the figure, while the outer sphere is defined with black lines. In the Torque editor, the inner sphere is made with red lines, and the outer sphere is made with blue lines.

The inner sphere represents the reference (or minimum) distance, and the outer sphere represents the maximum distance. The larger the outer sphere, the more gradual the drop-off in sound as you move away from the emitter. The larger the inner sphere, the farther the sound will carry.

Press F3 to switch to World Editor Inspector, and then click the hand cursor on the marker. At the bottom right, the editor frame contains the properties for the object, as show in Figure 20.12.

You can use this frame to adjust the settings for the emitter. Click the buttons in the Inspector frame to expand a selection of properties. After making changes, make sure to click the Apply button to have your changes applied to the selected object.

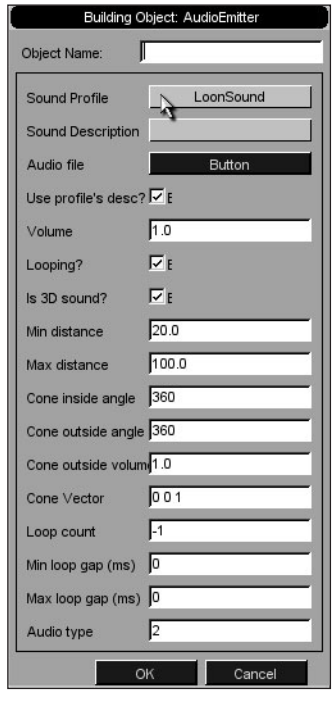


Figure 20.9 The Building Object: AudioEmitter dialog box.

Interface Sounds

Torque has a mechanism built in to offer sound effects when you use buttons. Objects that use the `GuiDefaultProfile`

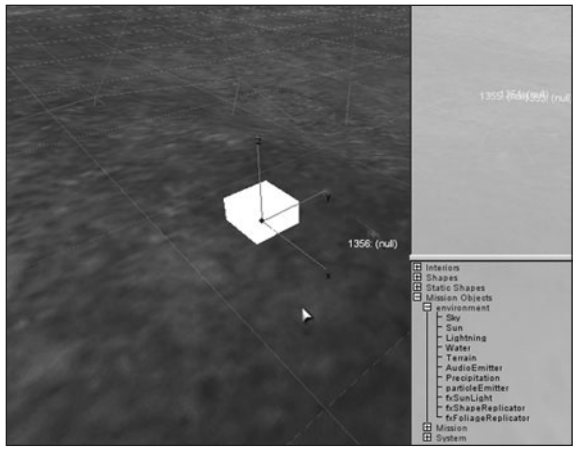


Figure 20.10 The AudioEmitter marker.

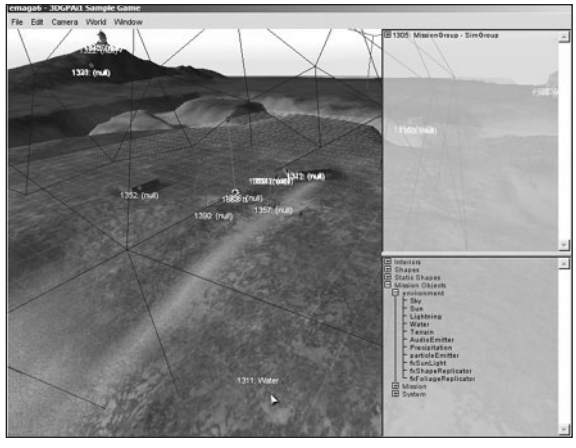


Figure 20.11 The AudioEmitter spheres.

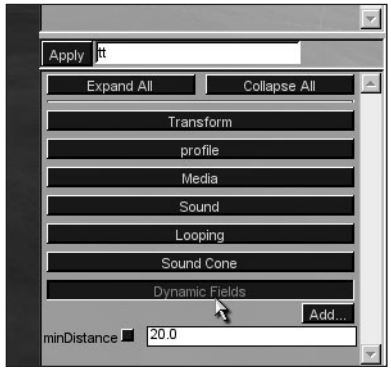


Figure 20.12 World Editor Inspector frame.

profile have two sound effects available: `soundButtonDown` and `soundButtonOver`.

If you look in C:\koob\control\client\default_profiles, you will find a sound effect for the buttonover context that I've created. This sound occurs whenever the cursor passes over a GUI button that has been defined to use the `GuiDefaultProfile`. The line looks like this:

```
GuiButtonProfile.soundButtonOver =
"AudioButtonOver";
```

It points the property to an `AudioProfile`. The profile is located in C:\koob\control\client\initialize.cs. The profile is located there and not in the sndprofiles.cs module because I need it to load before the menu with the buttons comes up. The AudioButtonOver profile looks like this:

```
new AudioProfile(AudioButtonOver)
{
   filename = "~/data/sound/but-
tonOver.wav";
   description = "AudioGui";
   preload = true;
};
```

It's pretty straightforward. A useful exercise for you at this point would be to create a sound effect for pressing down on the button, insert the appropriate audio profile code, and then point the `soundButtonDown` property at it, just as I showed you for the `soundButtonOver` property.

Music

You can handle music in much the same way as the simple sound effect I showed you at the beginning of

this chapter. A useful way to employ music is to provide a background for the different dialog boxes or menu screens in the GUI. Of course, you can also insert music into the game as AudioEmitters or even attach it to vehicles or players.

We'll take a slightly more conventional approach and put in some start-up music. First, locate the file C:\3DGPAi1\RESOURCES\CH20\TWLOGO.WAV and copy it over to C:\koob\control\data\sounds\.

Next, open the file C:\koob\control\client\misc\initialize.cs and add the following code to the top of the file:

```
new AudioDescription(AudioMusic)
{
   volume   = 0.8;
   isLooping= false;
   is3D     = false;
   type     = $MusicAudioType;
};

new AudioProfile(AudioIntroMusicProfile)
{
   filename = "~/data/sound/twlogo.wav";
   description = "AudioMusic";
   preload = true;
};
function PlayMusic(%handle)
{
   if (!alxIsPlaying(%handle))
     alxPlay(%handle);
}

function StopMusic(%handle)
{
   if (alxIsPlaying(%handle))
     alxStop(%handle);
}
```

Now scroll down a bit until you find this line:

```
   SetNetPort(0);
```

Add the following after it:

```
   StartMusic(AudioIntroMusicProfile);
```

This will start the opening music playing shortly after the first window appears once the game application has been launched.

Now open the file C:\koob\control\client\client.cs and insert the following line as the first line in the `LaunchGame()` function:

```
StopMusic(AudioIntroMusicProfile);
```

This line will ensure that if the opening music is still playing when you actually go to start the game, it will be turned off.

Now go ahead and launch the game and listen to the music.

You can use the same technique in combination with the `CommandToClient(`*`XXXX`*`)/clientCmd`*`XXXX`* system that we've used in earlier chapters to have the server trigger music cues on all or selected clients whenever you want.

Moving Right Along

There you go, enough sound that the people around you will be pestering you to turn the blasted game down!

You've seen the ways that sounds can be added for player-avatars, vehicles, and weapons. You've seen what a state machine does and how it helps define what sounds occur, and when, when using a weapon.

Then there's the ability to hurl insults at other players—a very important feature to know how to put into a game!

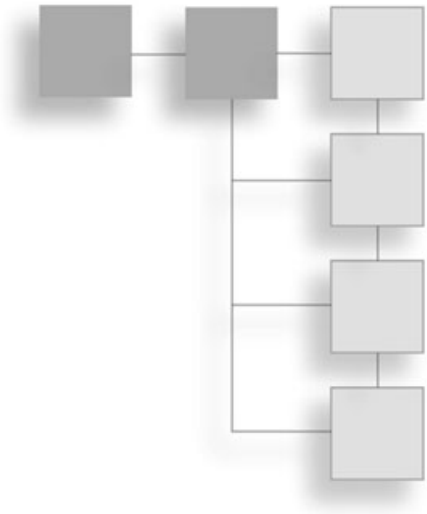
You've seen how to add sounds into your game world at specific locations, so that you can bring life to a babbling brook, or a howling wind on an open plain.

Adding sounds to the user interfaces, like buttons on interface screens, is really reasonably simple, as you've seen.

Finally, adding music to your game is really no more complex than any other sound, and in some cases, easier. You've seen how you can control the playing of music using Torque Script.

In the next chapter, we begin rolling all the things covered in earlier chapters together, by starting to create a game world.

CHAPTER 21



Creating the Game Mission

Let's take a moment to step back and see where we are. In the first chapters of the book you learned the basics of programming and how to apply those concepts to real things that can be done with a modern game engine. In the process you learned how to use a programming editor—UltraEdit—and how to use Torque to try out our ideas. You saw that Torque has a powerful script system and the things you can do with it are almost limitless.

We then moved on to artwork, starting with textures and graphic images. You learned about a couple of new tools—Paint Shop Pro and UVMapper—and how to apply them to the task of skinning 3D objects and GUI screens.

Then we got into modeling 3D objects, using a few more tools—MilkShape and QuArK—to create the models using different techniques. Animating our objects using MilkShape helped bring static models to life.

After that, it was on to some nifty things like creating skyboxes and images for weather effects, such as lightning and rain.

Then it was sound effects, recording them and using them in a game in different ways.

Now you might think that all of this is leading somewhere! And you would be right.

Game Design

Okay, enough typing and programming for now. It's time to examine some higher-level issues, like game design.

Team LRN

Start with a vision.

You have an idea. It can be an amorphous, gee-I'm-not-really-sure-but-something-like-this idea, or it can be concrete, specific, and detailed. Using that as your reference, start asking yourself questions, write them down, and don't worry about answering them…yet.

Requirements

There are a myriad of questions you can ask yourself when considering the design of your game:

- What will the genre or play style be like?
- Will this be a single-player or multiplayer game?
- If it is to be a multiplayer game, will it be an online game or a split-screen type of multiplayer game?
- Does the game relate to real-world activities?
- Does the player play as a creature (human, animal, alien, and so on) character or as some sort of machine?
- If the player isn't a creature or a machine, is he some sort of higher-level being that directs or controls a multitude of game entities?
- What are the player's goals?
- How do we measure player success and what are the scoring mechanisms?
- What are the challenges that the game presents to us as players?
- Will the challenges be designed (planned by the game developers) or will they be random?
- What is the backstory—the narrative that describes the world the game takes place in, if any?
- What is it about the game that will make people want to play it?
- What is the one skill or skill set that the game requires the player to master in order to succeed?
- What other skills will contribute to player success?
- What mechanisms can the game offer that will help players develop those skills?
- How much game enhancement of skills is too much help?

Feel free to add more questions of your own.

As you can see the list is big, and we've only scratched the surface here. By answering these questions, and any others you may want to add, we can build a list of requirements. It is important to generate at least this list—the *requirements specification*—in order to know where we are going with our design and how to measure our progress toward completion.

Team LRN

Software design is a Big Ticket Item, and hundreds of books have been written about it and the various design methodologies that have been proposed. It's an industry unto itself. There are dozens of different ideas about the best approach to take—and much disagreement. The one area *everyone* seems to agree on, however, is the need for the requirements to be determined and recorded in a meaningful way. Functional specs, test plans, schedules, and the like may or may not work for you, but you will get nowhere fast if you don't know what it is you are trying to accomplish.

Some of the questions lead to other questions. Some answers may need to be deferred until later when you have more information. Even if your list of questions exceeds your list of answers, it is still an important activity. Keep the list nearby, return to it regularly, and update the answers. See where it leads you. Perhaps you can see that you are wandering away from your original vision. The list may uncover things you'd never considered before, that *really are* important, and prevent you from wasting time on a mistaken approach.

When you work your way through creating the questions, try hard to stay general—steer away from specifics until they can't be avoided any longer.

At some point you will want to ask yourself, "What technology should I use to create this game?" Don't ask this question at the very beginning. In most cases you should wait until you know the answers to the bulk of your questions—in other words, wait until your requirements are starting to look meaty and useful—before you ask yourself technology questions.

Constraints

We usually have to accept that there are constraints that can arbitrarily force us to move in particular directions or prevent us from moving in others.

The design should drive the technology and not the other way around. However, in a low-budget development shop, this is often an unaffordable luxury. There are budgetary constraints caused by available funds that will force us in certain technological directions. In our case, because this book is about making games with minimal expenditure and using the Torque Game Engine to help us achieve that goal, we will have to accept that constraint and monitor the effect it has on our design as we build it up.

Again in our case, because we already know the technology we are going to use, we should examine it for its limitations and measure our constraints starting there.

As mentioned elsewhere Torque is designed for online, multiplayer, first-person shooter-style games. This means that whenever implementation tradeoffs had to occur when creating the game engine, the developers always tried to make their decisions in a way that favored efficient and fast networking, first-person perspective 3D rendering, and multiplayer support.

Team LRN

One of things that Torque, right out of the box, doesn't address in its design is massive multiplayer support. Torque can easily handle 64 players logged in to one session. It can even handle more than 100. In fact, there is no hard-coded limit to the number of players that could log in to the same server. But because of its design, Torque really starts to lose its shine when you exceed the realm of about 100 players.

Massively multiplayer games require the ability to have thousands of players playing the same game together. Torque just doesn't handle this kind of load.

So the server load is a constraint. While Torque's ability to handle 100 players at once is better than most, if not all, other FPS-style games out there, that still doesn't translate to thousands. We must keep this in mind.

The tools we have available can dictate other constraints. It's fine to decide that you will have a certain feature, but it may require an expensive 3D modeling tool to create and thus sit out of reach. So make sure you can create the things you want.

Koob

So let's go about listing some requirements for Koob, the game we are making. Feel free to add your own, but the list of 29 items here will serve as a starting place.

1. First-person and third-person perspective play.
2. Internet multiplayer game play.
3. Global in-game chat.
4. Ability to use at least one weapon.
5. Ability to get in and out of vehicles.
6. Ability to drive vehicles.
7. A road or track in the world to drive on.
8. Trees and other foliage.
9. Powerups: health, energy, ammunition, coins (for points).
10. Buildings that serve as hiding places and storage locations for powerups.
11. All other players are enemies.
12. All point values configurable from a setup file.
13. 1 point per enemy killed.
14. 3 points per vehicle destroyed.
15. Ability to race around the track and score 5 points for each lap you lead.
16. Laps can only be scored in the car.
17. A 10-lap race with no time limit.
18. A 10-point bonus for winning the race.

19. On-screen scorecard.
20. Scores retained at the end of each race restored when player resumes the game.
21. Each player gets an account at startup and must use a password to log in to the game.
22. Track must be clearly marked on the terrain.
23. Checkpoints along the way to measure progress and ensure the player stays on course.
24. Laps can only be completed when all checkpoints are completed for that lap.
25. Checkpoints must be completed in sequential order.
26. Coins of three denominations will be randomly scattered around the maps. The values will be 1 point, 10 points, and 100 points for copper, silver, and gold, respectively.
27. Some nice burning objects to admire as we play.
28. A waterfall to drive through just for the heck of it.
29. When one map is finished, cycle to the next in the list.

So as you can probably gather, Koob is a kind of death match scavenger hunt game. The player tries to win the race, accumulate some loot, and, at the same time, stop his enemies from winning.

As we progress from here to the end of the book, we'll check back against this list of requirements to make sure we've covered all the items.

Right off the bat, we can check off item 1. A few of the other items are definitely doable because we've chosen the Torque Engine to create the game, but we have some programming yet to do to make them happen.

Torque Mission Editor

You've already been exposed to the Mission Editor—a little bit here, a little bit there. As you've seen, the Mission Editor contains several subeditors: the World Editor, Terrain Editor, Terrain Terraformer Editor, Terrain Texture Editor, and Mission Area Editor. The main point of this section is to place objects in the game world and adjust them as required. To do this we will use the World Editor, which has two components: the World Editor Creator and its partner, the World Editor Inspector.

In the Mission Editor the normal movement keys can be used to control both the player and the camera. The right mouse button is used to rotate the camera or adjust the player's view.

Disk and file operations are carried out using the items in the File menu, as shown in Table 21.1. These include opening, saving, importing, and exporting.

Table 21.1 File Menu Commands

Command	Description
New Mission	Creates a new empty mission with a default terrain and sky.
Open Mission	Opens an existing mission for editing.
Save Mission	Saves changes to the current mission to disk.
Save Mission As	Saves the current mission under a new name.
Import Terraform Data	Imports terraform rules from an existing terrain file.
Import Texture Data	Imports terrain texture rules from an existing terrain file.
Export Terraform Bitmap	(Only active from the Terrain Terraform Editor.) Exports the current terraform map to a bitmap.

As is standard with windowed applications these days, there is an Edit menu that houses a variety of object and item editing commands. As you can see in Table 21.2, in addition to the ubiquitous Cut, Copy, and Paste functions, there are also commands that are used to access various settings for the editors.

Table 21.2 Edit Menu

Menu Item	Description
Undo	Undoes the last action in terrain or world editing. Not all actions can be undone.
Redo	Redoes the last undone action.
Cut	Cuts the selected objects in the World Editor from the mission to the Clipboard.
Copy	Copies the selected objects in the World Editor to the Clipboard.
Paste	Pastes the current clipboard contents into the mission.
Select All	Selects all mission objects in the World Editor.
Select None	Clears the current selection in the World and Terrain Editors.
Relight Scene	Recomputes the mission's static lighting and applies it.
World Editor Settings	Accesses the settings dialog box for the World Editor.
Terrain Editor Settings	Accesses the settings dialog box for the Terrain Editor.

Use the Camera menu, as described in Table 21.3, to change camera modes and adjust the camera fly mode speed.

The World menu is available by default and contains functions related to the World Editor—its capabilities will be described in the "World Editor" section, which is coming up next.

The Window menu is pretty straightforward, so it doesn't require a table to describe its functions. It is used to invoke each of the available subeditors.

Table 21.3 Camera Menu

Menu Item	Description
Drop Camera At Player	Moves the camera object to the location of the player, and sets the mode to camera movement mode (*camera fly mode*).
Drop Player At Camera	Moves the player object to the location of the movable camera, and sets the mode to player movement mode (*player mode*).
Toggle Camera	Toggles between player and camera fly movement modes. Your view will also switch to the location of either the player or the camera, depending on the mode.
Slowest to Fastest	Adjusts the movement speed of the camera fly mode.

World Editor

The World Editor provides a view of the 3D world. Objects in this view, like structures, interiors, shapes, and markers, can be manipulated with either the mouse or the keyboard.

There are three frames in the view: the World Editor Tree, the World Editor Inspector, and the World Editor Creator.

World Editor Tree

The World Editor Tree view is displayed in the frame in the upper-right screen corner in both the World Editor Inspector and the World Editor Creator. This tree displays the hierarchy of the mission data file. Objects selected in the Tree view will also be selected in the main view. Objects in the Tree view can be organized into groups.

There is a special group selection called the *Instant Group*, which is displayed with gray highlighting in the Tree view. This is the group in the Tree view where newly created or pasted objects are placed. Objects created from the World Editor Creator are also placed in the Instant Group. To change the current Instant Group, Alt+Click on a group in the Tree view.

World Editor Inspector

The World Editor Inspector lets you examine and specify properties of mission objects. When you select an object in Inspector mode, that object's properties are displayed in the frame at the lower right of the screen. After editing an object's properties, click the Apply button to commit those properties to the object. Dynamic properties can be assigned to objects with the Dynamic Fields Add button. Dynamic fields can be accessed via the scripting language and are normally used to add game-specific properties to objects.

World Editor Creator

The World Editor Creator displays an extra Tree view frame in the lower-right corner of the screen. This view contains all objects that can be created in a mission. Selecting an

object from this list creates a new instance of it and drops the new object at the center of the screen (by default) or as specified by the selected Drop at command in the World menu, which is shown in Table 21.4.

Table 21.4 World Menu

Menu Item	Description
Lock Selection	Locks the current selection so that it cannot be manipulated from the World Editor view.
Unlock Selection	Unlocks a locked selection.
Hide Selection	Hides the current selection to help reduce visual clutter.
Show Selection	Unhides hidden objects in the selection.
Delete Selection	Deletes the currently selected objects.
Camera To Selection	Moves the camera to the selected objects.
Reset Transforms	Resets the rotation and scale on the selected objects.
Drop Selection	Drops the selected objects into the mission according to the drop rule (see Drop Selection menu items that follow). If the object is already placed, it is picked up and dropped again. **Drop at Origin:** Drops newly created objects at the origin. **Drop at Camera:** Drops newly created objects at the camera's location. **Drop at Camera w/ Rot:** Drops newly created objects at the camera's location with the camera's current orientation. **Drop below Camera:** Drops newly created objects below the camera's current location. **Drop at Screen Center:** Drops newly created objects where the view direction hits an object. **Drop at Centroid:** Drops newly created objects at the center of the selection. **Drop to Ground:** Drops newly created objects to the terrain ground level at their current location.

You can use both the mouse and the keyboard for editing, as shown in Table 21.5.

Gizmos are the visual representation of each object's three axes. When you select an object, and if you have gizmos enabled in the World Editor Settings dialog box, then they will appear centered on that object's local origin.

If gizmos are enabled, they can be clicked and dragged (as described in Table 21.6) in order to modify the object to which they are attached.

Terrain Editor

We use the Terrain Editor to manually modify the terrain height map and square properties by using a mouse-operated brush. The brush is a selection of terrain points or squares

Table 21.5 Mouse and Keyboard Operations

Operation	Description
Clicking on an unselected object	Deselects all currently selected objects and selects the clicked object.
Clicking in empty space	Click-drags a box around objects, and selects all objects in the box.
Shift-clicking on an object	Toggles selection of the clicked object.
Mouse dragging a selected object	Moves the selected objects, either on a horizontal plane or sticking to the terrain, depending on the setting of the Planar Movement check box in the World Editor Settings dialog box.
Control-clicking and drag	Moves the selected objects vertically.
Alt-clicking and drag	Rotates the selected objects about the vertical axis.
Alt-Ctrl-clicking and drag	Scales the selected object by a face on the bounding box.

Table 21.6 Gizmo Operations

Operation	Description
Click-drag gizmo axis	Moves selection along the selected axis.
Alt-click-drag gizmo axis	Rotates selection on the selected axis.
Alt-Ctrl-click-drag gizmo axis	Scales along the selected axis.

centered around the mouse cursor. Table 21.7 describes the functions available in the Brush menu.

Table 21.7 Terrain Editor: Brush Menu

Menu Item	Description
Box Brush	Uses a square-shaped brush.
Circle Brush	Uses a circular brush.
Soft Brush	Sets the brush so that its influence on the terrain diminishes toward the edges of the brush. The brush square colors vary from red, where the influence is greatest, to green, where the influence is least. The Terrain Editor Settings dialog box Filter view has controls that adjust the falloff.
Hard Brush	Sets the brush so that the effect on the terrain is the same across the surface of the brush. All squares in the brush are the same red color.
Size 1×1 to 25×25	Sets the brush sizes.

When we use the Terrain Editor, we modify the terrain as if we were piling dirt onto it or shoveling holes into the ground. Table 21.8 shows the operations available in the Terrain Editor via the Action menu.

Team LRN

Table 21.8 Terrain Editor: Action Menu

Menu Item	Description
Select	Moves the brush in a painting motion to select grid points.
Adjust Selection	Raises or lowers the terrain at the currently selected grid points as a group by dragging the mouse up or down.
Add Dirt	Adds terrain "dirt" to the terrain at the center of the brush, raising the affected terrain area.
Excavate	Removes dirt from the center of the brush.
Adjust Height	Raises or lowers the area marked by the brush by dragging the mouse.
Flatten	Sets the area marked by the brush to a flat plane height.
Smooth	Smoothes the area marked by the brush—peaks are lowered and troughs are raised.
Set Height	Sets the area marked by the brush to a constant height—the height is set using the Terrain Editor Settings.
Set Empty	Makes a hole in the terrain in the squares covered by the brush.
Clear Empty	Fills in any holes in the squares covered by the brush.
Paint Material	Paints the current terrain texture material with the brush.

Terrain Terraform Editor

The Terrain Terraform Editor uses mathematical algorithms to generate terrain heightfields (height maps). Heightfield operations are arranged in a *stack*, which is an ordered list of operations. Operations in the stack depend on the results of previous operations to produce new heightfields. The results of the final operation on the stack can be applied to the terrain using the Apply button.

There are two Terrain Terraform Editor frames. The top frame displays information about the currently selected operation, and the bottom frame shows the current operation stack. Between them is a pull-down menu for the creation of new operations. The first operation in the stack is always the General operation, which can't be deleted.

Table 21.9 shows the operations available.

Terrain Texture Editor

The Terrain Texture Editor uses mathematical techniques to place terrain textures based on the heightfield at the bottom of the terraformer heightfield stack. The editor has three main interface elements on the right side of the screen. From top to bottom they are the operation Inspector frame, the Material list, and the Placement Operation list.

Terrain materials are textures that are added using the Add Material button. This will look for any texture (.png or .jpg) in a subdirectory of any directory named "terrains" (in this book, this *also* applies to a directory name maps). Once a material is added to the terrain,

Table 21.9 Terraform Operations

Operation	Description
fBm Fractal	Creates bumpy hills.
Rigid Multifractal	Creates ridges and sweeping valleys.
Canyon Fractal	Creates vertical canyon ridges.
Sinus	Creates overlapping sine wave patterns with different frequencies useful for making rolling hills.
Bitmap	Imports an existing 256 by 256 bitmap as a heightfield.
Turbulence	Jumbles the effects of another operation on the stack.
Smoothing	Smoothes the effects of another operation on the stack.
Smooth Water	Smoothes water.
Smooth Ridges/Valleys	Smoothes an existing operation on edge boundaries.
Filter	Applies a filter to an existing operation based on a curve.
Thermal Erosion	Applies an erosion effect to an existing operation using a thermal erosion algorithm.
Hydraulic Erosion	Applies an erosion effect to an existing operation using a hydraulic erosion algorithm.
Blend	Blends two existing operations according to a scale factor and a mathematical operator.
Terrain File	Loads an existing terrain file onto the stack.

the user can select one of several placement operations that govern where that material will be placed on the terrain. They are shown in Table 21.10.

Table 21.10 Terrain Texture Editor Placement Operations

Operation	Description
Place by Fractal	Places the terrain texture randomly across the terrain based on a Brownian motion fractal operation.
Place by Height	Places the texture based on an elevation filter.
Place by Slope	Places the texture based on a slope filter.
Place by Water Level	Places the texture based on the water level parameter in the Terraform Editor.

Click the Apply button to commit the current texture operation list to the terrain file.

Mission Area Editor

The Mission Area Editor defines regions in the game that are used to constrain player travel. If we use mission areas in a game, we normally give warnings or disqualifications if a player leaves a mission area. Of course, you can probably find other uses for such a feature.

Team LRN

The Mission Area Editor displays an overhead height-map view of the current mission map in the upper-right corner of the screen. There are markers for mission objects, a box for the mission area, and a pair of lines denoting the current field of view. Clicking anywhere on the display will move the current view object (either camera or player) to that location in the mission.

To edit the mission area, select the Edit Area check box. This will display eight resizing knobs on the mission area box that can be dragged with the mouse.

Clicking the Center button will cause the terrain file data to be repositioned and centered at 0,0 in the center of the mission area box.

To mirror the terrain, click the Mirror button. This will put the Mission Area Editor in mirror mode. The Left and Right Arrow buttons adjust the mirror plane angle to one of eight different angles (two axis aligned, two 45-degree splits). Click the Apply button to commit the terrain mirroring across the mirror plane. Mirroring a mission area is a useful way to quickly create terrain for team-based games where each side would begin with identical terrain. This would stop either side from having a terrain advantage. You create the terrain for one side, and then simply mirror it for the other side.

Building the World

Let's get to work building the game world, and let's start with items 27 and 28 from our requirements list. I've chosen the fire and the waterfall to start with here because we haven't really looked at particles much yet, and with particles we get to touch on various topics we've covered in this chapter, like easing into using the Mission Editor, the World Editor Creator, and the World Editor Inspector. And besides that, particles are cool.

Particles

Remember the raindrops in Chapter 18? Those were particles. *Particles* are basically single-faced polygons that are generated in bulk by a game engine to simulate a variety of somewhat related real-world phenomena, such as rain, smoke, wispy fog, splashing and spraying water or mud, fire, and flames. Particles can be used to simulate any sort of constantly changing fluid- or gaslike entity. Even a swarm of mosquitoes can be generated using particles.

What I'll do in this section is show you how to use the Torque particle system to make a campfire and a waterfall.

Particles are made of three parts:

- **Particle.** The actual things we see.
- **Particle Emitter.** The thing that causes the particles to come into existence.
- **Particle Emitter Node.** The object that the emitter is attached to.

If you attach the word *data* to the end of each, and remove the spaces, you'll have the formal names of the datablocks that define those terms of the particle system:

```
ParticleData
ParticleEmitterData
ParticleEmitterNodeData
```

Particles can live in the game world in one of two ways: as freestanding particles or as attached particles. Freestanding particles are defined using all three of the datablocks just mentioned, while attached particles only require defining the `ParticleData` and the `ParticleEmitterData`. The nodes aren't needed, because we are attaching the particle to some other object that supports particles. The object classes that support particles are players, weapons, projectiles, and all vehicle types. As noted already, `Rain` is a specialized object that has a built-in particle capability.

So in the case of freestanding particle emitters, one more definition of interest is required for placing emitters in the world:

```
ParticleEmitterNode
```

We'll look at freestanding particle emitters a bit more, shortly.

Campfire

To make a campfire, we'll need two particle definitions: one for the flames and one for the smoke. The particle types used will be freestanding, so we will need to define all three particle datablocks for both the smoke and the flames.

First, copy the image file C:\3DGPAi1\RESOURCES\CH21\flame.png to C:\koob\control\data\particles.

Next, create the file C:\koob\control\server\misc\particles.cs and add the following code to it:

```
datablock ParticleData(Campfire)
{
   textureName          = "~/data/particles/flame";
   dragCoefficient      = 0.0;
   gravityCoefficient   = -0.35;
   inheritedVelFactor   = 0.00;
   lifetimeMS           = 580;
   lifetimeVarianceMS   = 150;
   useInvAlpha = false;
   spinRandomMin = -15.0;
   spinRandomMax = 15.0;
```

```
   colors[0]     = "0.8 0.6 0.0 0.1";
   colors[1]     = "0.8 0.65 0.0 0.1";
   colors[2]     = "0.0 0.0 0.0 0.0";

   sizes[0]      = 1.0;
   sizes[1]      = 2.0;
   sizes[2]      = 4.0;

   times[0]      = 0.1;
   times[1]      = 0.4;
   times[2]      = 1.0;
};

datablock ParticleEmitterData(CampfireEmitter)
{
   ejectionPeriodMS = 15;
   periodVarianceMS = 5;

   ejectionVelocity = 0.35;
   velocityVariance = 0.20;

   thetaMin         = 0.0;
   thetaMax         = 60.0;

   particles = "Campfire" TAB "Campfire";
};

datablock ParticleEmitterNodeData(CampfireEmitterNode)
{
   timeMultiple = 1;
};
```

Now open C:\koob\control\server\server.cs, locate the function `OnServerCreated`, and add the following line to the end of the function, before the closing brace ("}"):

```
   exec("./misc/particles.cs");
```

Now, open your mission file (C:\koob\control\data\maps\koobA.mis or whatever your mission file is called, as long as it is the same as the one used for Chapter 6 and uses the same terrain) and add the following before the closing brace of the file:

```
   new ParticleEmitterNode() {
      position = "13.2665 -2.0218 196.6";
      rotation = "1 0 0 0";
```

```
      scale = "1 1 1";
      dataBlock = "CampfireEmitterNode";
      emitter = "CampfireEmitter";
      velocity = "1";
   };
```

Okay, save your files, and then launch Koob. When your player spawns in, turn to the right—you should see a little fire burning in the gully there, as shown in Figure 21.1.

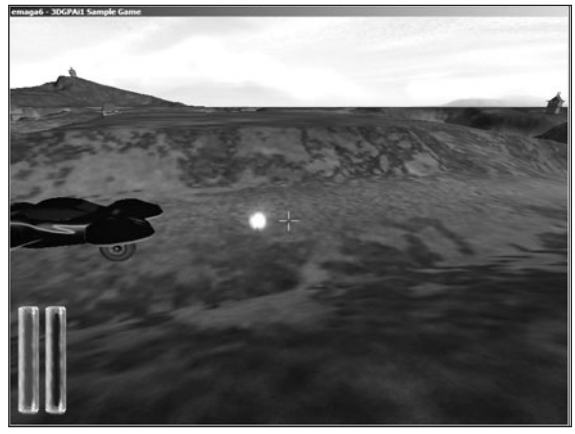


Figure 21.1 Campfire.

The flame is the glowing object to the left of the crosshair in the picture. Now let's add the smoke. We'll do it slightly differently. We begin by defining the particle and emitter as before, but then we'll place it in an easier way using the World Editor.

Open up the particles.cs file you created earlier, and add the following:

```
datablock ParticleData(CampfireSmoke)
{
   textureName          = "~/data/particles/smoke";
   dragCoefficient      = 0.0;
   gravityCoefficient   = -0.15;
   inheritedVelFactor   = 0.00;
   lifetimeMS           = 4000;
   lifetimeVarianceMS   = 500;
   useInvAlpha = false;
   spinRandomMin = -30.0;
   spinRandomMax = 30.0;
   colors[0]     = "0.5 0.5 0.5 0.1";
   colors[1]     = "0.6 0.6 0.6 0.1";
   colors[2]     = "0.6 0.6 0.6 0.0";
   sizes[0]      = 0.5;
   sizes[1]      = 0.75;
   sizes[2]      = 1.5;
   times[0]      = 0.0;
   times[1]      = 0.5;
   times[2]      = 1.0;
};
```

```
datablock ParticleEmitterData(CampfireSmokeEmitter)
{
   ejectionPeriodMS = 20;
   periodVarianceMS = 5;
   ejectionVelocity = 0.25;
   velocityVariance = 0.20;
   thetaMin         = 0.0;
   thetaMax         = 90.0;
   particles = CampfireSmoke;
};
datablock ParticleEmitterNodeData(CampfireSmokeEmitterNode)
{
   timeMultiple = 1;
};
```

Save your work and then launch Koob. Locate the campfire and face it in camera fly mode (press F8). Open the World Editor (press F11) and then enter the World Editor Creator (press F4). Browse the Tree view until you locate Mission Objects, environment, particleEmitter. Click it to place another particle emitter.

You will get the Building Object: ParticleEmitterNode dialog box. Using the illustration as a guide, choose CampfireSmokeEmitterNode from the datablock list, and then choose CampfireSmokeEmitter from the Particle data list (see Figure 21.2).

After the smoke appears, move it with the cursor until it's positioned directly over the campfire. Press F11 to get out of the editor, grab some s'mores, and get cookin'!

Figure 21.2 Adding smoke.

As you can see, `ParticleEmitterNodes` are useful for creating nodes that are stationary but animated. Place them in your world by adding a datablock and emitter references in your mission file, either through the Torque World Creator or by directly editing the mission file.

Table 21.11 describes the significant properties of the `ParticleEmitterNode` datablock. This describes the actual node object that is inserted in a mission file for a freestanding particle emitter.

Team LRN

Table 21.11 ParticleEmitterNode Properties

Property	Description
`velocity`	Acts as a master speed control modifying the settings for the `ParticleEmitterNodeData`, `ParticleData`, and `ParticleEmitterData` datablocks.
`datablock`	The name of the `ParticleEmitterNodeData` defined elsewhere that will be used.
`emitter`	The name of the `ParticleEmitterData` defined elsewhere that will be used.

Now, if you look at Figure 21.3, you'll see the relationship between the various datablocks involved in particles. The items in rectangles need to be defined somehow—you've seen how to do this. The one gotcha in the diagram is that the items in the dashed rectangles only need to be defined when placing freestanding particles in the game world. When you attach particles to objects like the player or vehicles, only the datablocks shown in the solid rectangles (`ParticleData`, and `ParticleEmitterData`) need to be defined.

Table 21.12 describes the significant properties of the `ParticleEmitterNodeData` datablock.

Table 21.12 ParticleEmitterNodeData Property

Property	Description
`timeMultiple`	Ranges from 0.01 to 100.0, specifying how often the particles are emitted from the node. Smaller values are for shorter time intervals between emissions, which means there is a higher emission frequency.

There is only one parameter in this datablock: `timeMultiple`. You can create any number of these datablocks with different settings and names.

Table 21.13 describes the significant properties of the `ParticleEmitterData` datablock.

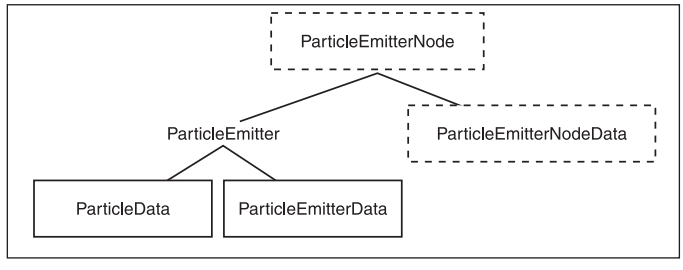


Figure 21.3 Particle system elements.

Table 21.13 ParticleEmitterData Properties

Property	Description
`ejectionPeriodMS`	Controls how often a particle is emitted in milliseconds (ms). A value of 1000 equals 1 particle per second (1 ms minimum).
`periodVarianceMS`	Introduces randomness to the ejection period. The variance must be less than `ejectionPeriodMS` and less than the `lifetimeMS` setting in the `ParticleData` section.
`ejectionVelocity`	Controls how fast the particle image is moved along the emission vector. Must be equal to or greater than 0, up to 3 meters per second maximum.
`velocityVariance`	Introduces randomness to the `ejectionVelocity`. The variance must be less than `ejectionVelocity`.
`ejectionOffset`	Modifies the start position of the particle ejection to occur at an offset along the ejection vector.
`thetaMax, thetaMin`	Sets the range (in degrees) for rotation around the X-axis of the `ParticleNode` object. `thetaMin` must be less than `thetaMax`, and both must be in the range of 0.0 to 180 degrees. The particle generator will randomly pick a value between those limits. Think of these properties together as "how high" the emitter's "aim" is.
`phiReferenceVel, phiVariance`	Sets the rotation angle around the Z-axis. Both arguments must be in the range of 0.0 to 180 degrees with `phiVariance` less than `phiReferenceVel`. Think of these properties together as the "direction" the emitter is pointing.
`overrideAdvances`	Defaults to `false`. When set to `true`, this will disable updating the particle as soon as it is created. This can be used to clean up particles generated by fast-moving objects.
`orientParticles`	Defaults to `false`. When set to `false`, the particle image is presented as a billboard that always faces the camera. When set to true, the particle image is oriented with respect to the ejection vector.
`orientOnVelocity`	Defaults to `false`. When set to `true`, the particle is displayed oriented with respect to the ejection vector. At the start the particle faces the screen, because velocity at the very beginning is 0.
`particles`	Contains the name of the `ParticleData` datablock to use. Multiple `ParticleData` datablocks can be specified in the string, separated by tab characters. The particle engine will cycle through the list repeatedly.
`lifetimeMS`	Defines how long this emitter will generate particles. It cannot be a negative value. A setting of 0 specifies no time limit. If not specified, then the default is 0.
`lifetimeVarianceMS`	Introduces randomness to the lifetime of the emitter. This value must be less than (and not equal to nor greater than) `lifetimeMS`.
`useEmitterSizes`	Does nothing if this datablock belongs to a `ParticleEmitterNode`. Otherwise, when set to `true`, use emitter-specified sizes instead of datablock sizes.
`useEmitterColors`	Does the same as `useEmitterSizes`, but for colors.

Team LRN

Table 21.14 describes the significant properties of the `ParticleData` datablock.

Table 21.14 ParticleData Properties

Property	Description
`textureName`	Specifies path and file name of a PNG or JPG image. Particle textures use black for the image areas that will be treated as the alpha (transparency) channel. PNG images will also use black for the transparent areas, but will also alternatively use the real alpha channel for transparent image regions if one is included. If a real alpha channel is specified in a PNG image, then black will not be used for transparency. Images must be sized in powers of 2, to a maximum of 512 by 512 pixels.
`useInvAlpha`	Switches from using black to using white for transparent regions.
`inheritedVelFactor`	Specifies how much of a parent object's velocity should be imparted in particles emitted.
`constantAcceleration`	Specifies acceleration rate for each particle along the ejection vector.
`dragCoefficient`	Specifies deceleration rate for each particle along the ejection vector.
`windCoefficient`	Specifies how much the game world's wind velocity vector should be imposed on particles emitted.
`gravityCoefficient`	Specifies acceleration rate for each particle vertically. Positive values indicate acceleration toward the ground.
`lifetimeMS`	Controls how long the particle image is displayed as it follows its ejection vector. Short lifetimes have a pronounced strobe effect. Default is 1000 (1 second) with a minimum value of 100.
`lifetimeVarianceMS`	Introduces randomness to the lifetime of the particle. This value must be less than (and not equal to nor greater than) `lifetimeMS`.
`spinSpeed`	Dictates how fast images will be randomly rotated around the vertical axis, if particles aren't set to be billboarded using the `orientParticles` or `orientOnVelocity` properties of the `ParticleEmitterData`.
`spinRandomMax`	Specifies the maximum allowable angle that a particle image can be randomly rotated. Allowable range is −10000.0 to +10000.0. `spinRandomMax` must be greater than `spinRandomMin`.
`spinRandomMin`	Specifies the minimum allowable angle that a particle image can be randomly rotated. Allowable range is −10000.0 to +10000.0. `spinRandomMin` must be less than `spinRandomMax`.
`animateTexture`	Allows use of animated particle image textures, when set to true.
`framesPerSec`	Specifies the animation frame rate.
`animTexName`	Specifies a DML file that contains a list of texture image files. Each file is a single frame in the animation.
`colors[`*n*`]`	Specifies the color interpolation values for three sequences of particle emissions.
`sizes[`*n*`]`	Specifies the scale interpolation values for three sequences of particle emissions.
`times[`*n*`]`	Specifies the time stamp values that pin the moments for the three particle emission sequences.

Team LRN

Waterfall

As promised, we will build a waterfall. Add the following particle system datablocks to your particles.cs file:

```
datablock ParticleData(WFallAParticle)
{
textureName = "~/data/particles/splash";
dragCoefficient = 0.0;
gravityCoefficient = 0.5;
windCoefficient = 1.0;
inheritedVelFactor = 2.00;
lifetimeMS = 15000;
lifetimeVarianceMS = 2500;
useInvAlpha = false;
spinRandomMin = -30.0;
spinRandomMax = 30.0;
colors[0] = "0.6 0.6 0.6 0.1";
colors[1] = "0.6 0.6 0.6 0.1";
colors[2] = "0.6 0.6 0.6 0.0";
sizes[0] = 5;
sizes[1] = 10;
sizes[2] = 15;
times[0] = 0.0;
times[1] = 0.5;
times[2] = 1.0;
};
datablock ParticleEmitterData(WFallAEmitter)
{
ejectionPeriodMS = 10;
periodVarianceMS = 5;
ejectionVelocity = 0.55;
velocityVariance = 0.30;
thetaMin = 0.0;
thetaMax = 90.0;
particles = WFallAParticle;
};
datablock ParticleEmitterNodeData(WFall1EmitterNode)
{
timeMultiple = 1;
};
//------------------------------
datablock ParticleData(WFallBParticle)
```

```
{
textureName = "~/data/particles/splash";
dragCoefficient = 0.0;
gravityCoefficient = -0.1; // rises slowly
inheritedVelFactor = 2.00;
lifetimeMS = 3000;
lifetimeVarianceMS = 500;
useInvAlpha = false;
spinRandomMin = -30.0;
spinRandomMax = 30.0;
colors[0] = "0.4 0.4 0.7 0.1";
colors[1] = "0.5 0.6 0.8 0.1";
colors[2] = "0.6 0.6 0.9 0.0";
sizes[0] = 10;
sizes[1] = 15;
sizes[2] = 20;
times[0] = 0.0;
times[1] = 0.5;
times[2] = 1.0;
};
datablock ParticleData(WFallCParticle)
{
textureName = "~/data/particles/splash";
dragCoefficient = 0.0;
gravityCoefficient = -0.1; // rises slowly
inheritedVelFactor = 2.00;
lifetimeMS = 3000;
lifetimeVarianceMS = 300;
useInvAlpha = false;
spinRandomMin = -30.0;
spinRandomMax = 30.0;
colors[0] = "0.4 0.4 0.5 0.1";
colors[1] = "0.5 0.5 0.6 0.1";
colors[2] = "0.0 0.0 0.7 0.0";
sizes[0] = 5;
sizes[1] = 5;
sizes[2] = 5;
times[0] = 0.0;
times[1] = 0.5;
times[2] = 1.0;
};
datablock ParticleEmitterData(WFallBParticleEmitter)
```

```
{
ejectionPeriodMS = 15;
periodVarianceMS = 5;
ejectionVelocity = 0.25;
velocityVariance = 0.10;
thetaMin = 0.0;
thetaMax = 90.0;
particles = "WFallBParticle" TAB "WFallCParticle";
};
datablock ParticleEmitterNodeData(WFall2ParticleEmitterNode)
{
timeMultiple = 1;
};
```

Save your work, and launch your game. The area where you want to make your waterfall is shown in Figure 21.4; the annotations show where I think is the best place for a waterfall. Anywhere along the water will do, though.

To get there, use F8 to get into camera fly mode, fly straight up for a second (do this by looking straight down at the ground and hitting the Down Arrow key to go backward and up). Then turn 180 degrees away from the direction you were facing when you spawned, and fly over to the area shown in the figure.

Once there, use the same methods used when you added the campfire smoke in the earlier section. For the top of the waterfall, use `WFall1ParticleEmitterNode` with the `WFallAEmitter`. Then for the splashing effect at the water surface, use `WFall2ParticleEmitterNode` with the `WFallBEmitter`. You should get something that looks like Figure 21.5.

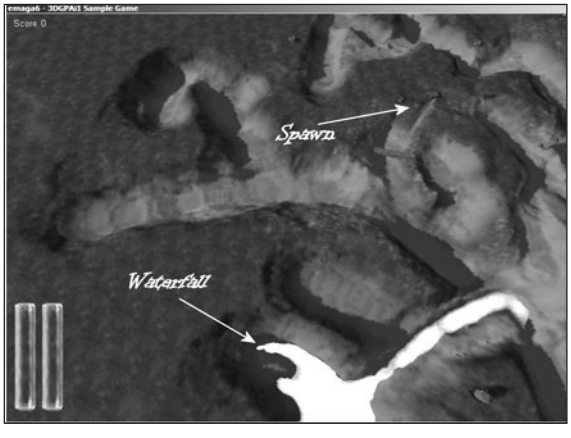


Figure 21.4 Locating the falls.

You can refine your waterfall by using three nodes for the top—one each for the left, right, and center of the falling water—and perhaps two at the bottom. You'll notice when you look at some waterfalls that the center stream of water has a different character than the outer fringes—hence the use of three particle emitters. Also, when the water hits the pool at the bottom, there are typically two observable phenomena: splashing and spraying of fine mist. So two emitters at the bottom would go a long way as well.

The Terrain

I've prebuilt two terrains—`trackA.ter` and `trackB.ter`—for use in Koob. Of course, you are free to make your own. Each of these terrains has a different, in fact a somewhat opposite, appearance.

`trackA.ter` is a bit claustrophobic in places, with much of the action happening in and around canyons and riverbeds, as you can see in Figure 21.6.

`trackB.ter`, in Figure 21.7, is more wide open, driving around a series of foothills and mountains.

In both cases, as I laid out the track, I wanted to make sure that there was no way to grossly cheat and find a shortcut that would allow someone who knew of the shortcut to obtain a huge advantage. In fact, I designed `trackB.ter` to have two built-in shortcuts that may or may not be quicker than staying on the track. See if you can find them!

Also, the specification that says that checkpoints must be used will help minimize shortcut use as cheating.

`trackA` is quite similar to the test terrain we've been using with Emaga6 and earlier revisions of Koob.

`trackB` is entirely new. If you want to check them out, then copy the files `trackA.mis`, `trackA.ter`, `trackB.mis`, and `trackB.ter` from C:\3DGPAi1\RESOURCES\CH21\ and deposit them in the C:\koob\control\data\maps\ directory.

Next, you will need to edit the file

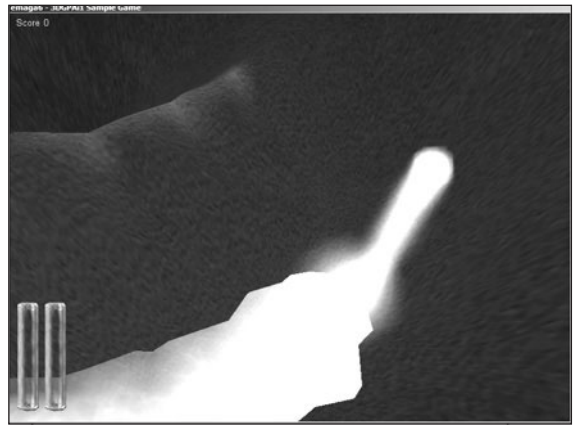


Figure 21.5 The falls.

Figure 21.6 trackA.ter.

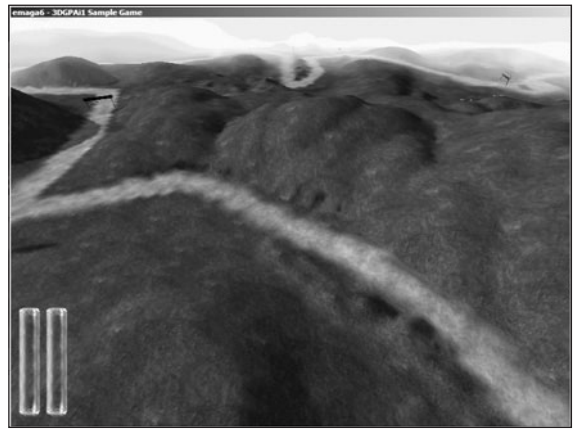


Figure 21.7 trackB.ter.

C:\koob\control\client\client.cs and find this line:

```
createServer("SinglePlayer", "control/data/maps/koobA.mis");
```

Then enter the file name of whichever mission file (and thus terrain) you want to look at. Now I know this seems to be an awkward way to select missions. We will be addressing this issue in the next chapters.

Items and Structures

Go ahead and set up Koob to use `trackB`. Once that's done, copy all of the files located in C:\3DGPAi1\RESOURCES\CH21\STRUCTURES and put them in C:\koob\control\data\maps\. If you get a dialog box asking if you want to overwrite existing files, your answer is yes.

Now launch the game and go into camera fly mode after you've spawned. You should be in the middle of a big parking lot.

Enter the World Editor Creator (press F11 followed by F4) and browse the Tree view until you find Interiors, Control, Data, Structures. Then look for the `startfinish` item and the `checkpoint` item, placing one of each in the game world, using Figure 21.8 as a guide.

To rotate an object, select it and then hold down the Alt key and hover the cursor over one of the gizmo axes (X, Y, or Z) of the item. When the axis label appears, click and hold. Then drag your cursor left or right, up or down, to cause the item to rotate around the chosen axis.

If you need to go back and adjust an object already placed, press F4 to enter the World Editor Inspector to select and adjust the items. Switch back to the World Editor Creator to resume placing items.

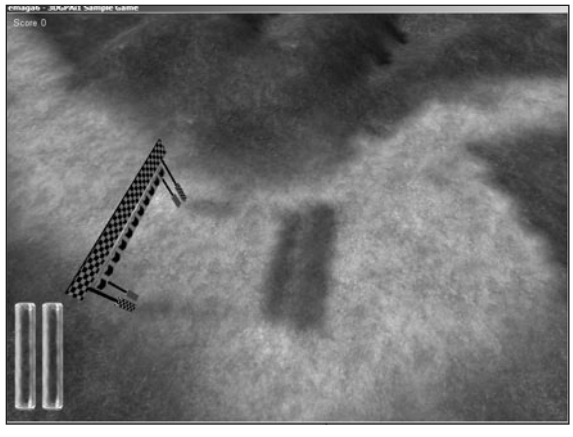


Figure 21.8 The start/finish line.

To move an item, select it, hover the cursor over one of the axis gizmos, and click and drag in the direction you want to move.

To scale an object along an axis, select the axis as before and then hold down both the Alt and Ctrl keys while dragging the cursor. The `checkpoint` object will have to be scaled a bit horizontally and vertically to fit it inside the `startfinish` object as depicted earlier in Figure 21.8.

You can use a number of other structures to define the track, like barriers (see Figure 21.9) and direction signs (see Figure 21.10).

You should place checkpoints at the locations indicated in Figure 21.11. There will be a total of five checkpoints: one at the start/finish line and four more around the track.

Place barriers strategically to prevent access to certain areas, and use the direction signs to assist players in understanding which direction the track will be heading.

You should also place a Tommy gun and crossbow accompanied by an ammo box for each somewhere in the vicinity of the start/finish line.

Place a Health Kit somewhere near the start/finish line as well. Also place a `HealthPatch` object near each checkpoint. You can use a block structure from the Interiors list to place these items on to improve visibility. Make sure to sink the block low enough into the ground that the player can jump up on it.

Also from the Interiors list, locate the hovels, and place a few around the track, near it but not too close. Make sure there is a way for a player to get to the hovels, and perhaps provide enough space to hide a vehicle behind them.

Select some trees and rocks from the Static Shapes list under Static Shapes, Control, Data, Models, Items. Place them around your map at visually appealing as well as strategic

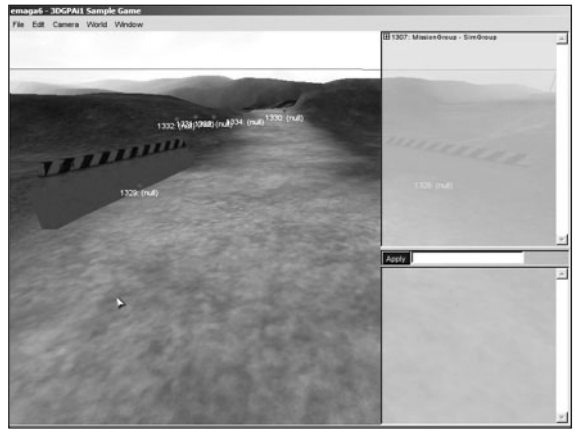


Figure 21.9 Barrier.

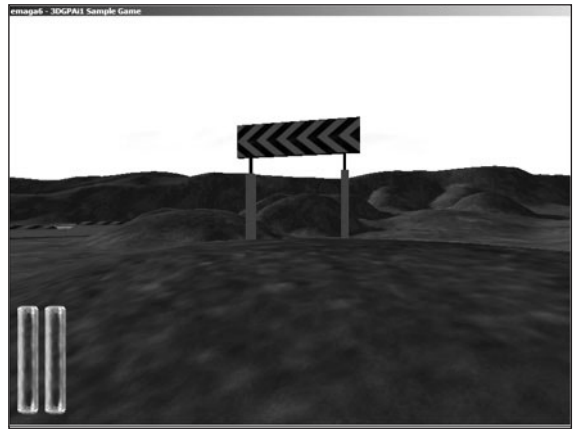


Figure 21.10 Direction sign.

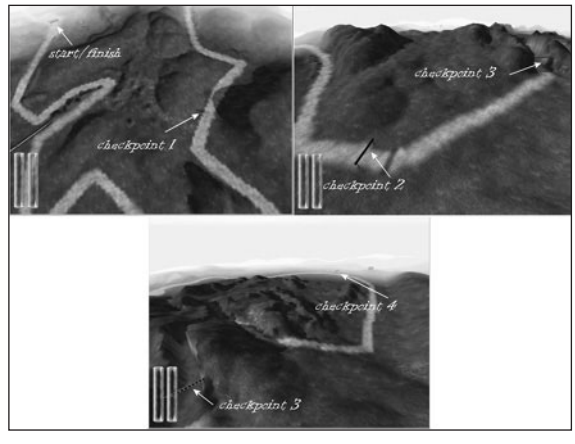


Figure 21.11 Checkpoint locations.

locations—you want to provide places for people to hide during an ambush. (Rocks to hide cars behind and so on.)

Go on and do the same sorts of things for `trackA`. There is a large flat area in a canyon near where you currently spawn (it's actually behind you), with a bridge leading across the river nearby. This is where you should put your start/finish line. The direction of travel will be to head from the start directly across the bridge and on from there. You probably won't need more than four checkpoints (other than the start/finish line) to complete the route. If you find you need to adjust the terrain to accommodate a building or something, by all means do it.

Don't worry about placing the coins. That is something we will handle with program code in the next chapter.

Moving Right Along

Well things got a little more hectic in the chapter. As you saw, designing a game is about answering questions. Often, the answers are the easy part—coming up with the questions can be tougher at times. Creating a requirements specification for your game is not only useful, it's almost a mandatory activity.

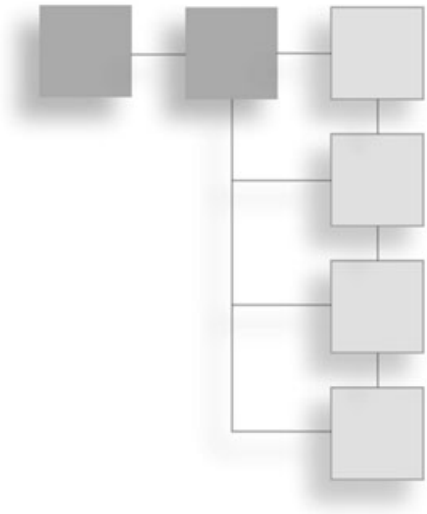
There are things that constrain our design, and we need to keep those constraints in mind. Every project will have different limits. One example you saw was that you probably shouldn't consider using Torque to make a massively multiplayer game.

We then looked at the Mission Editor in detail. You can use it to place and align all of the objects that will inhabit your game world. We used it to place some particle effects, in the form of a camp fire and a waterfall, as well as some structures that will be useful for gameplay.

Let's examine our requirements again. We checked off item 1 right from the get-go. Now we can check off a bunch of other items: 4, 5, 6, 7, 8, 10, 22, 27, and 28. We've also done parts of 9, 23, 24, and 25—but they need some programming as well to make them reality.

In the next chapter, we delve into more server-side gameplay issues, like spawning the player into random locations, getting a vehicle into the world, and triggering events.

CHAPTER 22



The Game Server

Now we have some things we've either added to the game world in recent chapters or simply included in our requirements that are not yet supported in the program code.

In this chapter we'll focus on adding the server-side code we need to support the requirements, as well as adding in some code to bring certain concepts to a more complete state.

The Player-Character

You've probably noticed a few things that are odd or incomplete in the player behavior or appearance in the code and art we've dealt with up to now. We'll tackle those things now.

Player Spawning

For our example games, we've used a fixed spawn point. Well, there is a convenient spawn point system available that we can employ.

First thing we need for this system is what we call a *marker*. Create a new file named C:\koob\control\server\misc\marker.cs, and add the following to it:

```
datablock MissionMarkerData(SpawnMarker)
{
   category = "Markers";
   shapeFile = "~/data/models/markers/sphere.dts";
};

function MissionMarkerData::create(%block)
{
```


Team LRN

```
   switch$(%block)
   {
      case "SpawnMarker":
         %obj = new SpawnSphere() {
            datablock = %block;
         };
         return(%obj);
   }
   return -1;
}
```

This has the by-now-familiar datablock, this one for a `MissionMarkerData`. The `Create` function tells the World Editor Creator how to make the new marker in the game world. Note the use of the `switch$` block, even though there is only one case—this is for later use for other kinds of markers. Save your work.

Now copy the directory C:\3DGPAi1\RESOURCES\CH22\markers and all of its contents to C:\koob\control\data\models\markers.

With that done, launch Koob, go into camera fly mode, and then move to a position overseeing the start/finish line, looking down at it. Go into World Editor Creator (press F11 followed by F4), and then add a `PlayerSpawns` group by choosing System, SimGroup from the Tree view at the lower right and entering `PlayerSpawns` as the object name in the dialog box. Make `PlayerSpawns` the current group by locating it in the hierarchy at the upper right and then holding down the Alt key while clicking the `PlayerSpawns` entry. The `PlayerSpawns` entry should now be highlighted in gray. Next, add a spawn marker by choosing Shapes, Markers, SpawnMarker from the Tree view at the lower right. A gray-white sphere will be placed in the world. Position about half a dozen or so of these around the start/finish area, hiding a few of them. Make sure they were all created in the `PlayerSpawns` group.

Now open the file C:\koob\control\server\server.cs and locate the function `SpawnPlayer`. Change the `createPlayer` call to look like this:

```
   %this.createPlayer(SelectSpawn());
```

Next, add the following method to the end of the file (or immediately after the `SpawnPlayer` function, if you like):

```
function SelectSpawn()
{
   %groupName = "MissionGroup/PlayerSpawns";
   %group = nameToID(%groupName);

   if (%group != -1) {
      %count = %group.getCount();
```

```
      if (%count != 0) {
         %index = getRandom(%count-1);
         %spawn = %group.getObject(%index);
         return %spawn.getTransform();
      }
      else
         error("No spawn points found in " @ %groupName);
   }
   else
      error("Missing spawn points group " @ %groupName);

   return "0 0 192 1 0 0 0"; // if no spawn points then center of world
}
```

This function will examine the `PlayerSpawns` group and count how many spawn markers are in it. It then randomly selects one of them and gets its transform (which contains the marker's position and rotation) and returns that value.

With this done, go ahead and try your game. Notice how each time you spawn, it's in a different place.

Vehicle Mounting

In recent chapters, when you've made your player-character get into the car, you may have noticed—especially from the third-person perspective—that the player is standing, with his head poking through the roof.

This is addressed by assigning values to a `mountPose` array. What we do is for each vehicle, we create `mountPoints` in the model (which we've done for the car). We need to specify in the car's model some nodes that will act as the mount points. We'll address the pose part of the player model in the next section, leaving the rest until the later section that covers vehicles.

The Model

In recent chapters I have been using a variation of the Standard Male Character as a filler for testing the code, maps, and other models. Now, however, it's time for you to use your own model, the Hero model we created back in Chapter 14. There are a few things we need to adjust in that model, so make a copy of your Hero model, and add him to your Koob models directory at C:\koob\control\data\models\avatars\hero. Create the hero directory if you haven't already done it. Copy all of your Hero model files, including the texture files, into that directory.

Adjusting Model Scale

You may find that your character is too large for your needs. If that is the case, it is easy to resolve. Make a judgment about how his size needs to change. Let's say he needs to be 50 percent bigger than he is now.

Fire up MilkShape and load your model. Look for a special material whose name starts with "opt". If the scale value in that name is 0.2, then change it to 0.3 (that's 1.5 times, or 150 percent of 0.2).

If you don't have a special material that adjusts the scale, create one. Switch to the Materials tab and create a new material and name it "opt:scale=0.15". By default, the DTS Exporter scales objects by 0.1, or one-tenth, the modeled size. So 0.15 is 150 percent of 0.1.

Animations

To properly mount your character in a vehicle, you will need to create a sitting pose. In MilkShape, add some more frames in the animation window—make sure to click the Anim button first!

Then select the last frame, and move the joints around until the character looks something like Figure 22.1.

Create a special material to be the sequence entry for this—it will be a one-frame sequence. Name the material "seq:sitting=102-102", save your work, and then export the file to your C:\koob\control\data\models\avatars\hero\ directory. The rest of the mounting stuff will be handled shortly in the "Vehicle" section.

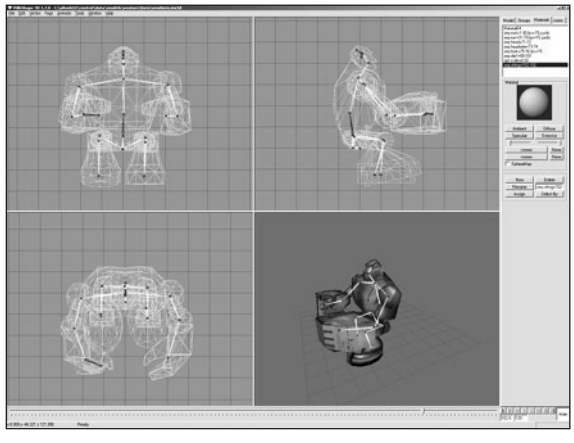


Figure 22.1 Sitting pose.

Server Code

Back in Chapter 20, I provided you with some code to mount and unmount the vehicle, just so you could hop in and out and test sounds. I didn't say much about what it did or why. Let's take a look now.

Collision

The premise is that you simply run up to a car and collide with it to get in. Now you must be mindful not to hit it too hard, or you will hurt yourself when

you get in. If you think that it shouldn't be so easy to hurt yourself, then you can edit your player's datablock to suit. Simply open C:\koob\control\server\players\player.cs and find the line that starts with `minImpactSpeed=` and increase the value —maybe to around 15 or so.

When your player collides with anything, the server makes a call, via the class name of your character's datablock to a callback method called onCollision.

tip

The class name for any datablock can be set via script like this:

```
classname = classname;
```

In the case of our player, *classname* is MaleAvatar, so the line is

```
classname = MaleAvatar;
```

Then methods are defined as

```
MaleAvatar::myMethod()
{ /// code in here
}
```

And they are invoked as

```
MyAvatarObjectHandle.myMethod();
```

`OnCollision` looks like this:

```
function MaleAvatar::onCollision(%this,%obj,%col,%vec,%speed)
{
  %obj_state = %obj.getState();
  %col_className = %col.getClassName();
  %col_dblock_className = %col.getDataBlock().className;
  %colName = %col.getDataBlock().getName();
  if ( %obj_state $= "Dead")
    return;
  if(%col_className $= "Item"||%col_className$="Weapon" )//Deal with all items
  {
    %obj.pickup(%col);                      // otherwise, pick the item up
  }
  if ( %col_className $= "WheeledVehicle" )
  {
      %node = 0;   // Find next available seat
      %col.mountObject(%obj,%node);
      %obj.mVehicle = %col;
  }
  else
```

```
  {
    %pushForce = %obj.getDataBlock().pushForce;  // Try to push the object away
    if (!%pushForce)
       %pushForce = 20;
    %eye = %obj.getEyeVector();          // Start with the shape's eye vector...
    %vec = vectorScale(%eye, %pushForce);
    %vec = vectorAdd(%vec,%obj.getVelocity());    // Add the shape's velocity
    %pos = %col.getPosition();                     // then push
    %vec    = getWords(%vec, 0, 1) @ " 0.0";
    %col.applyImpulse(%pos,%vec);
  }
}
```

In the parameters, `%this` refers to the datablock that this method belongs to, `%obj` is a handle to the instance of the avatar object that is our player in the game, `%col` is a handle to the object we've just hit, `%vec` is our velocity vector, and `%speed` is our speed.

The first thing we do is to check our object state, because if we are dead, we don't need to worry about anything anymore. We want to do this because dead avatars can still slide down hills and bang into things, until we decide to respawn. Therefore we need to stop dead avatars from picking up items in the world.

After that, we check the class name of the object we hit, and if it is an item that can be picked up, we pick it up.

Next, if the class is a `WheeledVehicle`, then this is where the mount action starts. The variable `%node` refers to the mount node. If `%node` is 0, then we are interested in the node `mount0`. That node is created in the model of the car, and the next section will show how we put that in. (This is not difficult—it's just a matter of creating a joint in the right place and naming it `mount0`.)

Then we make the call into the engine to `mountObject` for the car's object instance, and the game engine handles the details for us. We then update our player's instance to save the handle to the car we've just mounted.

If the object can't be picked up and is also not mountable, then we actually hit it. The next bit of code calculates our force based on our velocity and applies an impulse to the object we hit. So if we hit a garbage can, we will send it flying.

Mounting

Now when you call the `mountObject` method, the engine calls back to a method in the `ShapeBase` from which your avatar is derived. The method is `onMount`, and it looks like this:

```
function HumanMaleAvatar::onMount(%this,%obj,%vehicle,%node)
{
```

```
   %obj.setTransform("0 0 0 0 0 1 0");
   %obj.setActionThread(%vehicle.getDatablock().mountPose[%node]);
   if (%node == 0)
   {
      %obj.setControlObject(%vehicle);
      %obj.lastWeapon = %obj.getMountedImage($WeaponSlot);
      %obj.unmountImage($WeaponSlot);
   }
}
```

Now if you are wondering why the `onCollision` handler is accessed via the class name and the `onMount` handler is accessed via `ShapeBase`, I'll just have to admit that I'm not sure, and the answer isn't really that apparent. It's one of the vagaries of Torque, but if you keep it in mind, you won't have any problems.

The `onMount` method is interested in the `%obj`, `%vehicle`, and `%node` parameters. Our player is `%obj`, and obviously the vehicle we are mounting is `%vehicle`. The parameter `%node` refers to the mount node, as discussed earlier.

The first thing the code does is set our player to a null transform at a standard orientation, because the rest of the player object's transform information will be handled by the game engine, with the object *slaved* to the car—wherever the car goes, our player automatically goes as well.

Next, the mount pose is invoked, with the call to `setActionThread`. The animation sequence that was defined in the datablock as referring `mount0` is set in action. The animation sequence itself is only one frame, so the player just sits there, inside the car.

Now, if we are dealing with node 0, which by convention is always the driver, then we need to do a few things: arrange things so that our control inputs are directed to the car, save the information about what weapons we were carrying, and then unmount the weapon from our player.

Dismounting

Dismounting, or *unmounting*, is accomplished using whatever key is assigned to the jump action. It's a bit more involved than the mount code. First there is this bit:

```
function HumanMaleAvatar::doDismount(%this, %obj, %forced)
{
   %pos    = getWords(%obj.getTransform(), 0, 2);
   %oldPos = %pos;
   %vec[0] = " 1  1  1";
   %vec[1] = " 1  1  1";
   %vec[2] = " 1  1  -1";
   %vec[3] = " 1  0  0";
```

```
   %vec[4] = "-1  0  0";
   %impulseVec  = "0 0 0";
   %vec[0] = MatrixMulVector( %obj.getTransform(), %vec[0]);
   %pos = "0 0 0";
   %numAttempts = 5;
   %success     = -1;
   for (%i = 0; %i < %numAttempts; %i++)
   {
      %pos = VectorAdd(%oldPos, VectorScale(%vec[%i], 3));
      if (%obj.checkDismountPoint(%oldPos, %pos))
      {
         %success = %i;
         %impulseVec = %vec[%i];
         break;
      }
   }
   if (%forced && %success == -1)
      %pos = %oldPos;
   %obj.unmount();
   %obj.setControlObject(%obj);
   %obj.mountVehicle = false;
   %obj.setTransform(%pos);
   %obj.applyImpulse(%pos, VectorScale(%impulseVec, %obj.getDataBlock().mass));
}
```

Most of the code here is involved in deciding if the point chosen to deposit the player after removing him from the car is a safe and reasonable spot or not. We start by setting a direction vector, applying that vector to our player to figure out in advance where the proposed landing site for the freshly dismounted player will be, and then making sure it's okay using the `checkDismountPoint` method. If it isn't okay, the algorithm keeps moving the vector around until it finds a place that is suitable.

Once the site is determined, the `unMount` method is invoked and we return control back to our player model, deposit the model at the computed location, and give our player a little nudge.

When `unMount` is called, the game engine does its thing, and then it summons the callback `onUnmount`. What we do here is restore the weapon we unmounted.

```
function HumanMaleAvatar::onUnmount( %this, %obj, %vehicle, %node )
{
   %obj.mountImage(%obj.lastWeapon, $WeaponSlot);
}
```

Vehicle

We need to revisit the runabout model to prepare it for use as a mountable vehicle. The enhancement is not complex.

Model

Open the runabout model using MilkShape and add some mount nodes, as shown in Figure 22.2. These are joints, added using the Joint tool on the Model tab and named on the Joints tab.

Name the one in the driver's position `mount0` and the other `mount1`. Re-export your car and save it to C:\koob\control\data\models\vehicles\runabout.dts.

Earlier, when you mounted the vehicle using the filler model, the model was mounted to the local origin (0,0,0) of the car. Now the model will be mounted where you've specified with the nodes.

Datablock

We need to add a few things to the datablock. Open your file C:\koob\control\server\vehicles\car.cs and find the datablock. It looks like this:

```
datablock WheeledVehicleData(DefaultCar)
```

Add the following to the end of the datablock:

```
mountPose[0]              = "sitting";
mountPose[1]              = "sit-
ting";
numMountPoints            = 2;
```

The properties are pretty straightforward—`"sitting"` refers to the name of the sequence in the model that we created earlier with the Hero model in the sitting pose. The name was defined in a special material.

Table 22.1 contains descriptions of the most significant properties available for adjustment in the `WheeledVehicleData` datablock.

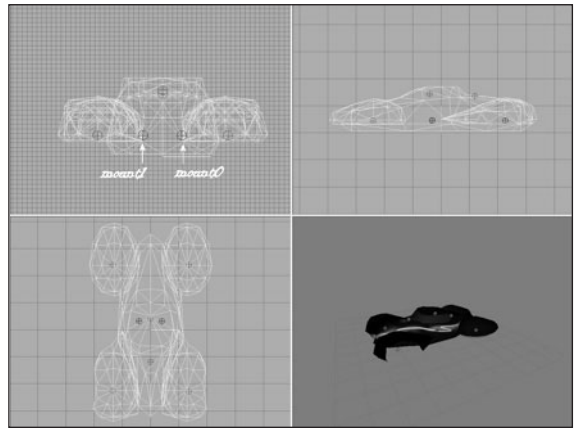


Figure 22.2 Car mount nodes.

Table 22.1 WheeledVehicleData Properties

Command	Description
`MaxDamage`	Specifies the maximum number of damage points a vehicle can take before it becomes disabled. Destroyed and disabled states are calculated percentages of this value.
`DestroyedLevel`	Specifies the percentage of `MaxDamage` that, when reached, causes the vehicle's `onDestroyed` callback to be called by the engine.
`DisabledLevel`	Specifies the percentage of `MaxDamage` that, when reached, causes the vehicle's `onDestroyed` callback to be called by the engine.
`MaxSteeringAngle`	Maximum steering angle.
`TireEmitter`	Same dust emitter as used by all the tires.
`CameraRoll`	Rolls the camera with the vehicle when it rolls.
`CameraMaxDist`	Farthest distance from the vehicle in third-person view.
`CameraOffset`	Vertical offset from the camera mount point.
`CameraLag`	Velocity lag of the camera in third-person view.
`CameraDecay`	Decay per second rate of velocity lag in third-person view.
`Mass`	Mass of the vehicle in quasi-kilograms.
`MassCenter`	Center of mass for rigid body expressed in object space 3D coordinates.
`MassBox`	Size of box used for moment of inertia; if 0 it defaults to the object's bounding box.
`Drag`	Drag coefficient. Used to counteract acceleration.
`BodyFriction`	Determines "stickiness" when the body brushes against the terrain or other objects.
`BodyRestitution`	Determines, by using rigid body physics, how much deformation is reversed.
`MinImpactSpeed`	Specifies the speed at and above which the vehicle's `onImpact` callback will be called by the engine.
`SoftImpactSpeed`	Specifies the speed at and above which the engine will play the vehicle's `SoftImpact` sound.
`HardImpactSpeed`	Specifies the speed at and above which the engine will play the vehicle's `HardImpact` sound.
`Integration`	Physics integration: TickSec/Rate. Higher values here yield higher integration. Higher integration leads to more accurate simulation but at the potential cost of CPU performance.
`CollisionTol`	Collision distance tolerance. A higher number means that a collision will be detected sooner (objects are farther apart) than with a lower number.
`ContactTol`	Contact velocity tolerance. How much leeway is allowed in determining whether objects have collided or have merely contacted, or brushed, each other. A higher number means that a more forceful contact can occur without the contact being considered a collision.
`EngineTorque`	Engine power, which causes acceleration, which leads to higher velocities.
`EngineBrake`	Braking when throttle is 0—simulates the internal "drag" of an engine that tends to slow a vehicle when it is in gear.
`BrakeTorque`	When brakes are applied, works as the opposite of `EngineTorque`.

continued

`MaxWheelSpeed`	The maximum rotation speed of the wheels, which directly affects the speed of the vehicle based on the wheel diameter and deformation factors. Wheel speed derives from engine speed and other factors.
`MaxEnergy`	The maximum amount of energy available to the vehicle for conversion into motion. Energy can be seen to be the same as fuel load.
`JetForce`	Additional boost force—a holdover term from the *Tribes* days. Means the same as acceleration.
`MinJetEnergy`	The smallest amount of energy needed to apply a jetting boost.
`JetEnergyDrain`	How quickly the energy of the vehicle is drained by use of jetting.
`JetSound`	The sound played when jetting or accelerating.
`EngineSound`	The sound played when the engine is idling.
`SquealSound`	The sound played when the tires skid.
`SoftImpactSound`	The sound played when a mild collision occurs.
`HardImpactSound`	The sound played when a serious collision occurs.
`WheelImpactSound`	The sound played when the wheels and tires hit something.

Two other datablocks have significant effect on the behavior of the car: `WheeledVehicleTire` and `WheeledVehicleSpring`, shown here:

```
datablock WheeledVehicleTire(DefaultCarTire)
{
   shapeFile = "~/data/models/vehicles/wheel.dts";
   staticFriction = 4;
   kineticFriction = 1.25;
   lateralForce = 18000;
   lateralDamping = 4000;
   lateralRelaxation = 1;
   longitudinalForce = 18000;
   longitudinalDamping = 4000;
   longitudinalRelaxation = 1;
};
datablock WheeledVehicleSpring(DefaultCarSpring)
{
   // Wheel suspension properties
   length = 0.85;              // Suspension travel
   force = 3000;               // Spring force
   damping = 600;              // Spring damping
   antiSwayForce = 3;          // Lateral anti-sway force
};
```

In the `WheeledVehicleTire` datablock you can see that tires act as springs in two ways: They generate lateral and longitudinal forces to move the vehicle. These distortion/spring forces are what convert wheel angular velocity into forces that act on the rigid body.

Team LRN

Triggering Events

When you need your players to interact with the game world, there is a lot that is handled by the engine through the programming of various objects in the environment, as we saw with collisions with vehicles. Most other interactions not handled by an object class can be dealt with using triggers.

A *trigger* is essentially a location in the game world, and the engine will detect when the player enters and leaves that space (*trigger events*). Based on the event detected we can define what should happen when that event is triggered using *event handlers* or *trigger callbacks*. We can organize our triggering to occur when there is an interaction with a specific object.

Creating Triggers

If you recall, some of our Koob specifications require us to count the number of laps completed. What we'll do is add a trigger to the area around the start/finish line, and every time a car with a player in it passes through this area, we'll increment the lap counter for that player.

For the trigger to know what object to call `onTrigger` for, you need to add an additional dynamic field with the name of instance of the trigger when it is created using the Mission Editor.

Open the file C:\koob\control\server\server.cs and at the end of the `onServerCreated` function, add this line:

```
exec("./misc/tracktriggers.cs");
```

This will load in our definitions.

Now create the file C:\koob\control\server\misc\tracktriggers.cs and put the following code in it:

```
datablock TriggerData(LapTrigger)
{
   tickPeriodMS = 100;
};

function LapTrigger::onEnterTrigger(%this,%trigger,%obj)
{
   if(%trigger.cp $= "")
      echo("Trigger checkpoint not set on " @ %trigger);
   else
      %obj.client.UpdateLap(%trigger,%obj);
}
```

The datablock declaration contains one property that specifies how often the engine will check to see if an object has entered the area of the trigger. In this case it is set to a 100-millisecond period, which means the trigger is checked 10 times per second.

There are three possible methods you can use for the trigger event handlers: `onEnterTrigger`, `onLeaveTrigger`, and `onTickTrigger`.

The `onEnterTrigger` and `onLeaveTrigger` methods have the same argument list. The first parameter, `%this`, is the trigger datablock's handle. The second parameter, `%trigger`, is the handle for the instance of the trigger object in question. The third parameter, `%obj`, is the handle for the instance of the object that entered or left the trigger.

In this `onEnterTrigger` the method is called as soon as (within a tenth of a second) the engine detects that an object has entered the trigger. The code checks the `cp` property of the trigger object to make sure that it has been set (not set to null or ""). If the `cp` property (which happens to be the checkpoint ID number) is set, then we call the client's `UpdateLap` method, with the trigger's handle and the colliding object's handle as arguments.

You can use `onLeaveTrigger` in exactly the same way, if you need to know when an object leaves a trigger.

The `onTickTrigger` method is similar but doesn't have the `%obj` property. This method is called every time the tick event occurs (10 times a second), as long as any object is present inside the trigger.

Next, we need to place the triggers in our world. We are going to put five triggers in, one at the start/finish line and one at each of the checkpoints.

Launch Koob, go into camera fly mode, and then move to a position overseeing (looking down at) the start/finish line. Go into the World Editor Creator (press F11 followed by F4), and then add a trigger by choosing Mission Objects, Mission, Trigger from the Tree view at the lower right.

Once you have your trigger placed, rotate and position it as necessary underneath the start/finish banner, and resize it to fill the width and the height of the area under the banner. Make the thickness roughly about one-tenth of the width, as shown in Figure 22.3.

Now switch to the World Editor (F3), locate your new object in the hierarchy at the upper right, and click it. In the Inspector frame, click the Dynamic Fields button, and then click Add. When you get the Add Dynamic Field dialog box (see Figure 22.4), enter "cp" in the name field and "0" in the value field, and click OK. Then click the Apply button to commit the changes to the object. What we've done is added a property to the object and named it "cp" with the value 0. We can access this property later from within the program code. The next checkpoint will be numbered 1, the one after that will be 2, next is 3, and finally 4, which is the fifth checkpoint. The numbering proceeds in a counterclockwise direction.

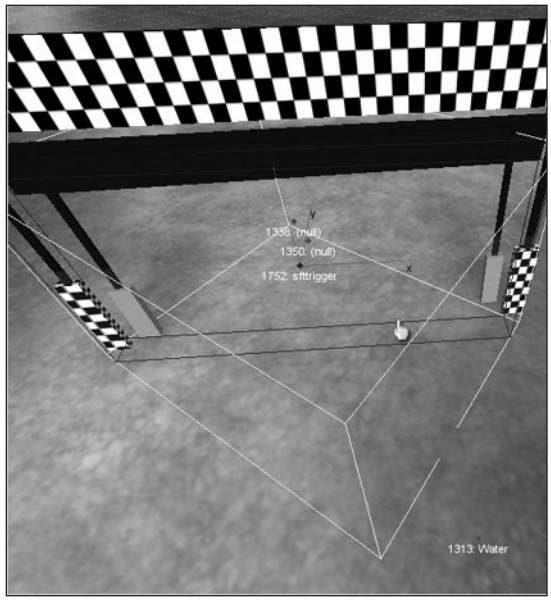


Figure 22.3 Placing a trigger.

Go ahead and add those checkpoints now, using the same technique as just noted. You can copy and paste the first trigger object to create the rest if you like—just remember to change the `cp` property accordingly.

tip

Some objects behave a little oddly when added via copy and paste. After pasting an object into the world, even though it will be visually selected in the view of the world, it still needs to be selected in the Inspector hierarchy in the upper-right frame. There are times when this may not be strictly necessary, but if you move, rotate, or resize the object by directly manipulating it via the gizmo handles, the changes will not be reflected in the Inspector frame until you reselect the object in the hierarchy.

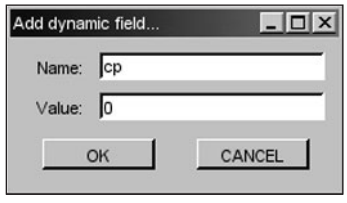


Figure 22.4 The Add Dynamic Field dialog box.

Now we have the ability to measure progress around the track. We have to add code to use these triggers, and that will be done as part of the scoring system, which is in the next section.

Scoring

We need to keep track of our accomplishments and transmit those values to the client for display.

Laps and Checkpoints

Open the file C:\koob\control\server\server.cs and put the following code at the end of the `GameConnection::CreatePlayer` method:

```
%client.lapsCompleted = 0;
%client.cpCompleted = 0;
%client.ResetCPs();

%client.position = 0;
%client.money = 0;
%client.deaths = 0;
%client.kills = 0;
%client.score = 0;
```

These are the variables we use to track various scores. Now add the following methods to the end of the file:

```
function GameConnection::ResetCPs(%client)
{
    for (%i = 0; %i < $Game::NumberOfCheckpoints; %i++)
      %client.cpCompleted[%i]=false;
}
function GameConnection::CheckProgress(%client, %cpnumber)
{
    for (%i = 0; %i < %cpnumber; %i++)
    {
      if (%client.cpCompleted[%i]==false)
        return false;
    }
    %client.cpCompleted = %cpnumber;
    return true;
}
function GameConnection::UpdateLap(%client,%trigger,%obj)
{
  if (%trigger.cp==0)
  {
    if (%client.CheckProgress($Game::NumberOfCheckpoints))
    {
        %client.ResetCPs();
        %client.cpCompleted[0] = true;
        %client.lapsCompleted++;
        %client.DoScore();
        if(%client.lapsCompleted >= $Game::NumberOfLaps)
          EndGame();
    }
    else
    {
      %client.cpCompleted[0] = true;
      %client.DoScore();
    }
  }
  else if (%client.CheckProgress(%trigger.cp))
  {
    %client.cpCompleted[%trigger.cp] = true;
    %client.DoScore();
  }
}
```

```
function GameConnection::DoScore(%client)
{
  %scoreString  =            %client.score          @
                   " Lap:" @ %client.lapsCompleted @
                   " CP:"  @ %client.cpCompleted+1 @
                   " $:"   @ %client.money          @
                   " D:"   @ %client.deaths         @
                   " K:"   @ %client.kills;
  commandToClient(%client, 'UpdateScore', %scoreString);
}
```

Starting from the last, the `DoScore` method merely sends a string containing scores to the client using the messaging system. The client code to handle this string will be presented in Chapter 23.

Before that is the meat of these particular functions—`UpdateLap`. You will recall that this is the method that is called for the client from the `onEnterTrigger` method.

The first thing `UpdateLap` does is to check to see if this is the first checkpoint, because it has a special case. Because we will start and drive through the first checkpoint at the start/finish line, it can be legitimately triggered without any other trigger events having occurred. We want to check for this condition. We check this by calling `CheckProgress` to see how many triggers have been passed. If the answer is none (a `false` return value), then we are starting the race, so we mark this checkpoint as having been completed and update our score to reflect that fact.

If this *isn't* the first checkpoint, then we want to check if all the checkpoints up until this checkpoint have been completed for this lap. If so, then mark this one completed and update the score; otherwise just ignore it.

Now finally, if we are back at checkpoint 0 and when we check to see if all the other checkpoints have been passed the result is `true`, then we are finishing a lap. So we increment the lap, reset the checkpoint counters, mark this checkpoint completed, update the score, and then check to see if the race is over—if not, we continue.

The previous method, `CheckProgress`, is called from `UpdateLap` and receives the current checkpoint ID number as a parameter. It then loops through the checkpoint array for this client and verifies that all lower-numbered checkpoints have been set to `true` (they have been passed). If any one of them is `false`, then this checkpoint is out of sequence and not legitimate. The function then returns `false`; otherwise all is in order, and it returns `true`.

And then first, but not least (grins), is the method `ResetCPs`. This simple method just riffles through the checkpoint array setting all entries to `false`.

Now there are a few odds and ends to deal with: Earlier in this file, server.cs, is the `StartGame` function. Locate it, and add these lines after the last code in there:

Team LRN

```
   $Game::NumberOfLaps = 10;
   $Game::NumberOfCheckpoints = 5;
```

Of course, you should adjust these values to suit yourself. You might want to set `NumberOfLaps` to a lower number, like 2, for testing purposes. Speaking of testing, if you want to test this, but without having addressed the client-side code first, then you can add some `echo` statements and view the output in the console window (invoked by pressing the Tilde key). A good place to put such a statement would be just before the `CommandToClient` call in `DoScore`. It would look like this:

```
echo( "Score " @ %scoreString  );
```

Money

Another requirement was to have randomly scattered coins in the game world.

Open C:\koob\control\server\server.cs and locate the function `StartGame` and add the following line to the end of the function:

```
   PlaceCoins();
```

Then place the following function just after the `StartGame` function:

```
function PlaceCoins()
{
%W=GetWord(MissionArea.area,2);
%H=GetWord(MissionArea.area,3);
%west = GetWord(MissionArea.area,0);
%south = GetWord(MissionArea.area,1);
  new SimSet (CoinGroup);
  for (%i = 0; %i < 4; %i++)
  {
    %x = GetRandom(%W) + %west;
    %y = GetRandom(%H) + %south;
    %searchMasks = $TypeMasks::PlayerObjectType | $TypeMasks::InteriorObjectType |
$TypeMasks::TerrainObjectType | $TypeMasks::ShapeBaseObjectType;
    %scanTarg = ContainerRayCast(%x SPC %y SPC "500", %x SPC %y SPC "-100", %search-
Masks);
    if(%scanTarg && !(%scanTarg.getType() & $TypeMasks::InteriorObjectType))
    {
      %newpos = GetWord(%scanTarg,1)  SPC GetWord(%scanTarg,2) SPC GetWord(%scanTarg,3)
+ 1;
    }
    %coin = new Item("Gold "@%i) {
      position = %newpos;
```

```
      rotation = "1 0 0 0";
      scale = "5 5 5";
      dataBlock = "Gold";
      collideable = "0";
      static = "0";
      rotate = "1";
    };
    MissionCleanup.add(%coin);
    CoinGroup.add(%coin);
  }
  // repeat above for silver coin
  for (%i = 0; %i < 8; %i++)
  {
    %x = GetRandom(%W) + %west;
    %y = GetRandom(%H) + %south;
    %searchMasks = $TypeMasks::PlayerObjectType | $TypeMasks::InteriorObjectType |
$TypeMasks::TerrainObjectType | $TypeMasks::ShapeBaseObjectType;
    %scanTarg = ContainerRayCast(%x SPC %y SPC "500", %x SPC %y SPC "-100", %search-
Masks);
    if(%scanTarg && !(%scanTarg.getType() & $TypeMasks::InteriorObjectType))
    {
      %newpos = GetWord(%scanTarg,1)  SPC GetWord(%scanTarg,2) SPC GetWord(%scanTarg,3)
+ 1;
    }
    %coin = new Item("Silver "@%i) {
      position = %newpos;
      rotation = "1 0 0 0";
      scale = "5 5 5";
      dataBlock = "Silver";
      collideable = "0";
      static = "0";
      rotate = "1";
    };
    MissionCleanup.add(%coin);
    CoinGroup.add(%coin);
  }
  // repeat above for copper coin
  for (%i = 0; %i < 32; %i++)
  {
    %x = GetRandom(%W) + %west;
    %y = GetRandom(%H) + %south;
    %searchMasks = $TypeMasks::PlayerObjectType | $TypeMasks::InteriorObjectType |
```

```
$TypeMasks::TerrainObjectType | $TypeMasks::ShapeBaseObjectType;
    %scanTarg = ContainerRayCast(%x SPC %y SPC "500", %x SPC %y SPC "-100", %search-
Masks);
    if(%scanTarg && !(%scanTarg.getType() & $TypeMasks::InteriorObjectType))
    {
      %newpos = GetWord(%scanTarg,1)  SPC GetWord(%scanTarg,2) SPC GetWord(%scanTarg,3)
+ 1;
    }
    %coin = new Item("Copper "@%i) {
      position = %newpos;
      rotation = "1 0 0 0";
      scale = "5 5 5";
      dataBlock = "Copper";
      collideable = "0";
      static = "0";
      rotate = "1";
    };
    MissionCleanup.add(%coin);
    CoinGroup.add(%coin);
  }
}
```

The first thing this function does is to obtain the particulars of the `MissionArea`. For this game, you should use the Mission Area Editor (press F11 followed by F5) to expand the `MissionArea` to fill the entire available terrain tile.

The `%H` and `%W` values are the height and width of the `MissionArea` box. The variables `%west` and `%south` combined make the coordinates of the southwest corner. We uses these values to constrain our random number selection.

Then we set up a search mask. All objects in the Torque Engine have a mask value that helps to identify the object type. We can combine these masks using a bitwise-or operation, in order to identify a selection of different types of interest.

Then we use our random coordinates to do a search from 500 world units altitude downward until we encounter terrain, using the `ContainerRayCast` function.

When the ray cast finds terrain, we add 1 world unit to the height and then use that plus the random coordinates to build a position at which to spawn a coin. Then we spawn the coin using the appropriate datablock, which can be found in your new copy of item.cs.

Next, we add the coin to the `MissionCleanup` group so that Torque will automatically remove the coins when the game ends. We also add it to the `CoinGroup` in case we want to access it later.

After putting that code in, copy C:\3DGPAi1\RESOURCES\CH22\ITEM.CS over to C:\koob\control\server\misc. You will find the datablocks for the coins (where the coin values are assigned) in there.

Note that when we added the coins in the preceding code, the `static` parameter was set to 0. This means that the game will not create a new coin at the place where the coin was picked up, if it is picked up. The weapons of the ammo do this, but we don't want our coins to do it. It's a game play design decision.

In addition to the datablocks for the coins in item.cs, you will also find this code:

```
   if (%user.client)
   {
      messageClient(%user.client, 'MsgItemPickup', '\c0You picked up %1', %this.pickup-
Name);
      %user.client.money += %this.value;
      %user.client.DoScore();
   }
```

The last two statements in there allow the player to accumulate the money values, and then the server notifies the client of the new score. Note that it is similar in that small way to the checkpoint scoring.

Again, until the client code is in place, you can insert `echo` statements there to verify that things are working properly.

Deaths

We want to track the number of times we die to further satisfy requirements, so open C:\koob\control\server\server.cs, locate the method `GameConnection::onDeath`, and add these lines at the end:

```
  %this.deaths++;
  %this.DoScore();
```

By now these lines should be familiar. We can expand the player death by adding some sound effects and animation. Add the following to the end of C:\koob\control\server\players\player.cs:

```
function Player::playDeathAnimation(%this,%deathIdx)
{
   %this.setActionThread("Die1");
}
datablock AudioProfile(DeathCrySound)
{
   fileName = "~/data/sound/orc_death.wav";
   description = AudioClose3d;
```

```
   preload = true;
};
function Player::playDeathCry( %this )
{
   %client = %this.client;
   serverPlay3D(DeathCrySound,%this.getTransform());
}
```

The first function, `playDeathAnimation`, will play the animation sequence we named `Die1` in our model. After that is another audio profile, pretty straightforward, followed by the function `playDeathCry`, which will play that profile's sound effect. These are invoked by two lines that you should place in the `OnDisabled` function farther back up in the player.cs file. Add these two lines to `OnDisabled` just before the call to `SetImageTrigger`:

```
%obj.playDeathCry();
%obj.playDeathAnimation(-1);
```

One more thing—copy the audio wave file C:\3DGPAi1\RESOURCES\CH22\ORC_DEATH.WAV to C:\koob\control\data\sound in order to make the sound work.

Kills

The victim, who notifies the shooter's client when he dies, actually does the kill tracking. So we go back to `GameConnection::onDeath` and add this:

```
%sourceClient = %sourceObject ? %sourceObject.client : 0;
if (%obj.getState() $= "Dead")
{
   if (isObject(%sourceClient))
   {
      %sourceClient.incScore(1);
      if (isObject(%client))
         %client.onDeath(%sourceObject, %sourceClient, %damageType, %location);
   }
}
```

This bit of code figures out who shot the player and notifies the shooter's client object of this fact.

Now it is important to remember that all this takes place on the server, and when we refer to the client in this context, we are actually talking about the client's *connection object* and not about the remote client itself.

Okay, so now let's move on to the client side and finish filling the requirements!

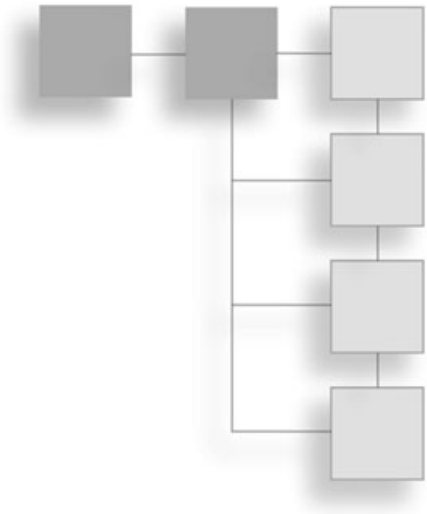
Moving Right Along

So, now we have our player's model ready to appear in the game as our avatar, we've got wheels for him to get around in, and a way to figure out where he's been.

We've also put some things in the game world for the player to pursue to accumulate points, and a way to discourage other players from accumulating too many points for themselves (by killing them).

All of these features are created on the server. In the next chapter, we will add the features that will be handled by the game client.

CHAPTER 23



The Game Client

By now we've met most of our requirements, at least to the point of implementation. Testing them for correct operation and completeness I will leave as an exercise for you, Gentle Reader, because you may (and probably will) want to modify and enhance the requirements anyway.

According to my list, the requirements that remain outstanding are the following:

2. Internet multiplayer game play.
3. Global in-game chat.
11. All other players are enemies.
12. All point values configurable from a setup file.
14. 3 points per vehicle destroyed.
15. Ability to race around the track and score 5 points for each lap you lead. (partial)
16. Laps can only be scored in the car.
17. A 10-lap race with no time limit.
18. A 10-point bonus for winning the race.
29. When one map is finished, cycle to the next in the list.

Of this list, I will leave numbers 14, 16, 17, and 18 and the remaining portion of number 15 (scoring 5 points) to you to complete as exercises. They are variations of the coin scoring and the lap and checkpoint tracking we covered in Chapter 22. The functioning code is available in the Koob installation kit on the CD, if you need help.

Most of the remaining work requires additional client code to support the server additions we made in the last chapter—we'll add some multiplayer support, a little bit more client support, and user interfaces to access those capabilities.

Team LRN

Client Interfaces

We are going to add code to allow users to run a server and to allow players to connect to a server. In order to make that connection, we will want to provide the user with an interface he can use to find servers, decide which one offers an interesting game, and then connect to the server.

Another thing we need to do is make sure that when the user quits a server, he returns to his selection interface rather than simply exiting as Koob does now.

Additionally, we need to add a capability to the playing interface to provide a chat window with a text entry where players can type in messages to send to other players. Maybe they'll want to exchange recipes or something. Yeah, that's it—recipes! It's not like they're going to taunt anyone anyway, is it?

In Chapter 6 you saw the `MasterScreen` interface module that combined these interfaces. In this chapter we'll look at the same issue but in a slightly different way, in order to show how easy it is to make different—yet equally valid—design decisions.

Also, we'll need to modify a few of the files, like the `MainScreen` interface, to more closely conform to our needs.

In a later section we'll add the code required to make these interfaces functional.

MenuScreen Interface

We will make some changes to our main menu screen so that it provides the user with the additional choices to

- view information about the games and credits
- play in single-player mode (as it already has)
- host a game
- connect to another server

Open your MenuScreen.gui file and locate the following line:

```
command = "LaunchGame();";
```

This line is a property statement in a `GuiButtonCtrl`. Delete the entire control, from where it says

```
   new GuiButtonCtrl() {
```

down to the closing brace ("}").

In the place of the deleted control, insert the following:

```
new GuiButtonCtrl() {
   profile = "GuiButtonProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "30 138";
   extent = "120 20";
   minExtent = "8 8";
   visible = "1";
   command = "Canvas.setContent(SoloScreen);";
   text = "Play Solo";
   groupNum = "-1";
   buttonType = "PushButton";
      helpTag = "0";
};
new GuiButtonCtrl() {
   profile = "GuiButtonProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "30 166";
   extent = "120 20";
   minExtent = "8 8";
   visible = "1";
   command = "Canvas.setContent(ServerScreen);";
   text = "Find a Server";
   groupNum = "-1";
   buttonType = "PushButton";
      helpTag = "0";
};
new GuiButtonCtrl() {
   profile = "GuiButtonProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "30 192";
   extent = "120 20";
   minExtent = "8 8";
   visible = "1";
   command = "Canvas.setContent(HostScreen);";
   text = "Host Game";
   groupNum = "-1";
   buttonType = "PushButton";
      helpTag = "0";
};
```

```
new GuiButtonCtrl() {
   profile = "GuiButtonProfile";
   horizSizing = "right";
   vertSizing = "top";
   position = "30 237";
   extent = "120 20";
   minExtent = "8 8";
   visible = "1";
   command = "getHelp();";
   helpTag = "0";
   text = "Info";
   groupNum = "-1";
   buttonType = "PushButton";
};
```

You may, if you wish, use the built-in GUI Editor (press F10) to do this. Make sure that you set all of the properties to match those just listed.

The significant thing to note about these controls is the `command` property. Each one replaces a displayed `MenuScreen` interface with a new interface, according to its function, with the exception of the Info button.

The Info button uses the `getHelp` feature of the common code base. It searches all of the directories nested under the root main directory looking for files with the extension .hfl, and then it lists them in alphanumeric order. If you preface the file name with a number, such as 1., 2., and so on, it will sort them numerically.

This should give you a main menu that looks like Figure 23.1.

Figure 23.1 MenuScreen interface.

SoloPlay Interface

The `SoloPlay` interface, as shown in Figure 23.2, prepares a list of mission files that it finds in the maps subdirectory in the control\data directory tree. From this list, you can select the map or mission you want to play. Its code and definition can be found in the SoloScreen modules.

It's worth remembering that even when you play in solo mode, underneath the hood, the Torque Engine is

still running in two parts: a client and a server. They are just closely coupled with no cross-network calls being made.

Host Interface

The `Host` interface is somewhat similar, as you can see in Figure 23.3, but offers more options: the ability to set a time limit and a score limit, plus map selection modes. Its code and definition can be found in the HostScreen modules.

If both time and score limits are set, the first one reached ends the game. A setting of 0 makes that limit infinite. The sequence mode causes the server to step through the maps in order as shown in the listing, as each game finishes and the new one loads. The random mode causes the server to randomly select a map for each game. The time limit is saved by the control in the variable `$Game::Duration`, and the score limit is saved as `$Game::MaxPoints`.

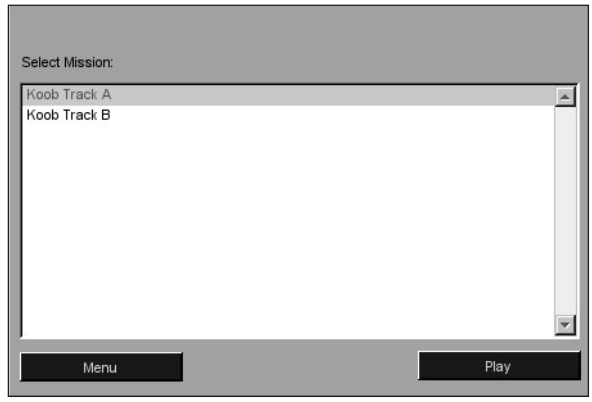


Figure 23.2 SoloPlay interface.

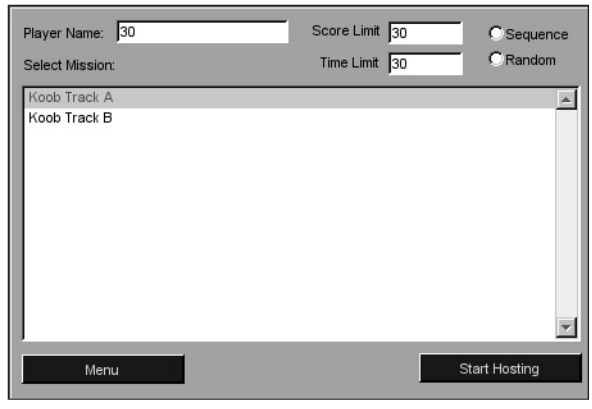


Figure 23.3 Host interface.

FindServer Interface

The `FindServer` interface, shown in Figure 23.4, lets you browse for servers. Its code and definition can be found in the ServerScreen modules. It will find servers that are running on the local LAN you are connected to (if you are connected to one, of course), and it will attempt to reach out via the Internet to contact the master servers at GarageGames and find games for you to connect to. You are not required to use the GarageGames master servers, but then you will have to write your own master server software to connect to. This can be done using Torque Script but is beyond the scope of this book. There are master server resources available from the GarageGames user community.

note

The Query LAN button on the `FindServer` interface does the same thing that was done automatically in Chapter 6 when you clicked on the Connect to Server button on the main menu screen. The discussion in the Chapter 6 section called ServerScreen Code Module describes how the Connect to Server button operations are performed, which is the same as how the Query LAN button operations work here.

ChatBox Interface

In order to display chat messages from other players, we need to put a control in our main player interface. We also need to have a control that will allow us to type in messages to be sent to other players, as depicted in Figure 23.5.

Open the file C:\koob\control\client\Initialize.cs and add the following lines to the function `InitializeClient`:

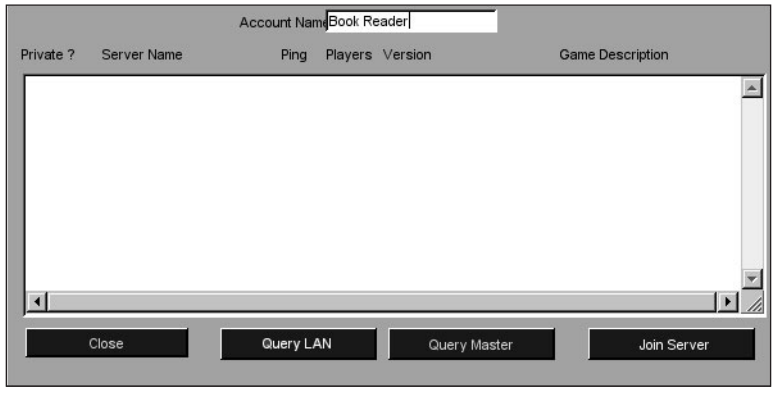


Figure 23.4 FindServer interface.

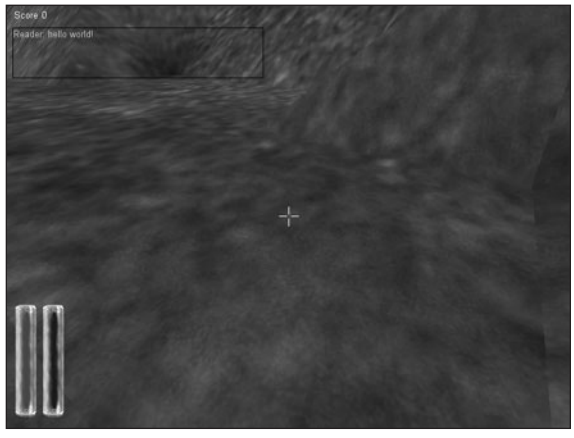


Figure 23.5 ChatBox interface.

```
   Exec("./interfaces/ChatBox.gui");
   Exec("./interfaces/MessageBox.gui");
```

These `exec` statements load the new files that will provide our chat interface. You can copy them from C:\3DGPAi1\RESOURCES\CH23 and put them into the directories under the C:\koob\control\client\ directory in the subdirectories specified in the `exec` statements.

Now open the file C:\koob\control\client\misc\presetkeys.cs and add the following keyboard input binding statements to the end of the file:

```
function pageMessageBoxUp( %val )
{
   if ( %val )
      PageUpMessageBox();
}
function pageMessageBoxDown( %val )
{
   if ( %val )
      PageDownMessageBox ();
}
PlayerKeymap.bind(keyboard, "t", ToggleMessageBox );
PlayerKeymap.bind(keyboard, "PageUp", PageMessageBoxUp );
PlayerKeymap.bind(keyboard, "PageDown", PageMessageBoxDown );
```

The first two functions are glue functions that are called by two of the key bindings at the bottom and then make the appropriate call to the functions that scroll the messages in the message box. We need these functions in order to filter out the key up and key down signals from the engine. We only want the action to take place when the key is pressed. We can do this by checking the value of `%val` when we enter the function—it will be nonzero when the key is pressed and zero when it is released.

Then there is a binding that calls `ToggleMessageBox`, which is defined in `MessageBox.cs` (one of the files we've recently copied that we will examine shortly).

In the interface files there are a couple of concepts you should note. To illustrate, look at the definition of the `ChatBox` interface, contained in ChatBox.gui:

```
new GuiControl(MainChatBox) {
   profile = "GuiModelessDialogProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "1";
   modal = "1";
```

```
setFirstResponder = "0";
noCursor = true;

new GuiNoMouseCtrl() {
   profile = "GuiDefaultProfile";
   horizSizing = "relative";
   vertSizing = "bottom";
   position = "0 0";
   extent = "400 300";
   minExtent = "8 8";
   visible = "1";

   new GuiBitmapCtrl(OuterChatFrame)
   {
      profile = "GuiDefaultProfile";
      horizSizing = "width";
      vertSizing = "bottom";
      position = "8 32";
      extent = "256 72";
      minExtent = "8 8";
      visible = "1";
      setFirstResponder = "0";
      bitmap = "./hudfill.png";

      new GuiButtonCtrl(chatPageDown)
      {
         profile = "GuiButtonProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "217 54";
         extent = "36 14";
         minExtent = "8 8";
         visible = "0";
         text = "Dwn";
      };

      new GuiScrollCtrl(ChatScrollFrame)
      {
         profile = "ChatBoxScrollProfile";
         horizSizing = "width";
         vertSizing = "bottom";
         position = "0 0";
```

```
            extent = "256 72";
            minExtent = "8 8";
            visible = "1";
            setFirstResponder = "0";
            willFirstRespond = "1";
            hScrollBar = "alwaysOff";
            vScrollBar = "alwaysOff";
            constantThumbHeight = "0";

            new GuiMessageVectorCtrl(ChatBox)
            {
               profile = "ChatBoxMessageProfile";
               horizSizing = "width";
               vertSizing = "height";
               position = "4 4";
               extent = "252 64";
               minExtent = "8 8";
               visible = "1";
               setFirstResponder = "0";
               lineSpacing = "0";
               lineContinuedIndex = "10";
               allowedMatches[0] = "http";
               allowedMatches[1] = "tgeserver";
               matchColor = "0 0 255 255";
               maxColorIndex = 5;
            };
         };
      };
   };
};
```

You've probably noticed that there is a heck of lot of indenting. This shows that there are many nested objects within objects. Each nesting level is there for a reason.

The outer level, owned by `MainChatBox`, is a general-purpose `GuiControl` container that encompasses the entire screen, occupying the same extents as the `Canvas` that we view the 3D world through.

Inside that is a `GuiNoMouseCtrl` control whose role is to shield the chat boxes within it from being accessible by a mouse cursor, if you were to display one on the screen.

Inside that is the `GuiBitmapCtrl` control named `OuterChatFrame`, which has two useful functions. You can use it to provide a nice bitmap background for your chat box if you want one, and it holds two subobjects.

Team LRN

One of those subobjects is an icon that appears to tell you when you've scrolled the chat box up far enough to hide text off the bottom of the box. That control is a `GuiButtonCtrl` named `chatPageDown`.

The other control is a `GuiScrollCtrl` named `ChatScrollFrame` that provides scroll bars for both vertical and horizontal scrolling.

And finally, in the inner sanctum is the actual control that contains the text of the chat box when it is displayed. This `GuiMessageVectorCtrl` supports multiline buffers of text that will display new text at the bottom and scroll older text up. You can use commands (that we have bound to the PageUp and PageDown keys) to scroll up and down through the text buffer.

MessageBox Interface

The `MessageBox` interface is where we type in our messages, as shown in Figure 23.6.

It is not normally on the screen but pops up when we hit the key we bound to it. This, too, has several nested levels, though not as many as the `ChatBox` interface.

```
new GuiControl(MessageBox)
{
   profile = "GuiDefaultProfile";
   horizSizing = "width";
   vertSizing = "height";
   position = "0 0";
   extent = "640 480";
   minExtent = "8 8";
   visible = "0";
   noCursor = true;
```

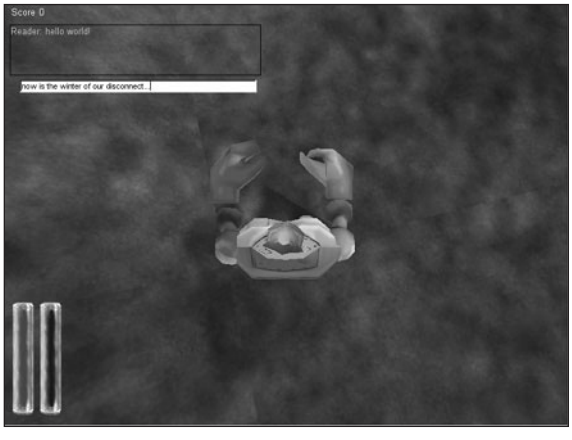


Figure 23.6 MessageBox interface.

```
   new GuiControl(MessageBox_Frame)
   {
      profile = "GuiDefaultProfile";
      horizSizing = "right";
      vertSizing = "bottom";
      position = "120 375";
      extent = "400 24";
      minExtent = "8 8";
      visible = "1";

      new GuiTextCtrl(MessageBox_Text)
      {
         profile = "GuiTextProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "6 5";
         extent = "10 22";
         minExtent = "8 8";
         visible = "1";
      };

      new GuiTextEditCtrl(MessageBox_Edit)
      {
         profile = "GuiTextEditProfile";
         horizSizing = "right";
         vertSizing = "bottom";
         position = "0 5";
         extent = "10 22";
         minExtent = "8 8";
         visible = "1";
         altCommand = "$ThisControl.eval();";
         escapeCommand = "MessageBox_Edit.onEscape();";
         historySize = "5";
         maxLength = "120";
      };
   };
};
```

It is all familiar stuff, but take note that the outer object, `MessageBox`, is initially invisible. The code that pops the box up will make it visible and invisible again as needed.

There is a `GuiTextCtrl` named `MessageBox_Text` that is at the same level as the `GuiTextEditCtrl` named `MessageBox_Edit`. The `MessageBox_Text` can be used to put a prompt in front of the area where the message will be typed in, although it has no text here in the

definition. The `MessageBox_Edit` control is the control that accepts our typed-in message. The `altCommand` property specifies what statement to execute when the Enter key is pressed, and the `escapeCommand` property specifies what to do when the Escape key is pressed. The handlers for these two functions will be discussed later in the code discussion in the "Client Code" section.

Client Code

I'm not going to make you type in great reams of program code at this stage of the game, though you don't get off the hook completely. You will have to make some changes to accommodate the new stuff, and we'll also examine the contents of some of the new stuff to see what it does.

MenuScreen Interface Code

Open the file C:\koob\control\client\initialize.cs and locate the function `InitializeClient` and add the following lines in the grouping with the other similar statements:

```
Exec("./misc/ServerScreen.cs");
Exec("./misc/HostScreen.cs");
Exec("./misc/SoloScreen.cs");
Exec("./interfaces/ServerScreen.gui");
Exec("./interfaces/HostScreen.gui");
Exec("./interfaces/SoloScreen.gui");
```

Like I promised, I won't make you type in all the files referenced in those `exec` statements; you can copy them from C:\3DGPAi1\RESOURCES\CH23 and put them into the directories under the C:\koob\control\client\ directory in the subdirectories specified in the `exec` statements.

Each of these files is basically one module split into two parts. The actual interface definitions are in the files with the .gui extensions, while the code that manages the interfaces are in the files with the same prefix name, but ending with the .cs extension.

If you go back to the previous code listing for MenuScreen.gui, you will see where each of the interfaces is invoked. `ServerScreen` is defined in ServerScreen.gui, `HostScreen` is defined in HostScreen.gui, and finally, `SoloScreen` is defined in SoloScreen.gui.

Each interface has roughly the same form. There is an `OnWake` method for the interface object that is called by the engine when that object is displayed by the `SetContent` call in the related button in the `MenuScreen` interface. This method prepares the interface and fills the various data fields in the interfaces.

SoloPlay Interface Code

The `SoloPlay` interface that you saw in Figure 23.2 prepares a list of mission files that it finds so that you can select one of them to play with. The functional code for the `SoloPlay` interface, extracted from SoloPlay.cs, is shown here for discussion:

```
function PlaySolo()
{
   %id = SoloMissionList.getSelectedId();
   %mission = getField(SoloMissionList.getRowTextById(%id), 1);
   StopMusic(AudioIntroMusicProfile);
   createServer("SinglePlayer", %mission);
   %conn = new GameConnection(ServerConnection);
   RootGroup.add(ServerConnection);
   %conn.setConnectArgs("Reader");
   %conn.connectLocal();
}

function SoloScreen::onWake()
{
   SoloMissionList.clear();
   %i = 0;
   for(%file = findFirstFile($Server::MissionFileSpec);
         %file !$= ""; %file = findNextFile($Server::MissionFileSpec))
      if (strStr(%file, "CVS/") == -1 && strStr(%file, "common/") == -1)
         SoloMissionList.addRow(%i++, getMissionDisplayName(%file) @ "\t" @ %file );
   SoloMissionList.sort(0);
   SoloMissionList.setSelectedRow(0);
   SoloMissionList.scrollVisible(0);
}
function getMissionDisplayName( %missionFile )
{
   %file = new FileObject();
   %MissionInfoObject = "";
   if ( %file.openForRead( %missionFile ) ) {
      %inInfoBlock = false;

      while ( !%file.isEOF() ) {
         %line = %file.readLine();
         %line = trim( %line );

         if( %line $= "new ScriptObject(MissionInfo) {" )
            %inInfoBlock = true;
```

```
         else if( %inInfoBlock && %line $= "};" ) {
            %inInfoBlock = false;
            %MissionInfoObject = %MissionInfoObject @ %line;
            break;
         }

         if( %inInfoBlock )
               %MissionInfoObject = %MissionInfoObject @ %line @ " ";
      }

      %file.close();
   }
   %MissionInfoObject = "%MissionInfoObject = " @ %MissionInfoObject;
   eval( %MissionInfoObject );
   %file.delete();
   if( %MissionInfoObject.name !$= "" )
      return %MissionInfoObject.name;
   else
      return fileBase(%missionFile);
}
```

The `OnWake` method is as described in earlier chapters—in this case the `onWake` method makes clear the mission list and then populates it according to the matching files it finds in the path indicated by `$Server::MissionFileSpec`. This variable is set in the file C:\koob\control\server\initialize.cs with the following line in the `InitializeServer` function:

```
$Server::MissionFileSpec = "*/maps/*.mis";
```

There are a couple of things you should understand about the way the search is done in the code presented.

First, there is the matter of the syntax used here. It can be difficult to decipher C-based code because of the looseness allowed—and Torque Script's syntax is extremely close to that of the C language and C++. You will recall that with most statements that employ a code block, such as `if` and `for`, you can use the long form or the short form, depending on your needs.

For example, the long form using braces

```
if (%a==1) { %x=5; }
```

can also be written as

```
if (%a==1) {
  %x=5;
}
```

or as

```
if (%a==1)
{
  %x=5;
}
```

There are also other minor variations, but I'm sure you get the idea. The compiler doesn't care about the lines the code appears on, and it doesn't care about the amount of *white space* (tabs, spaces, and carriage returns). It only cares that the correct tokens and keywords are in the right place and that they make sense according to the compiler's parsing rules. Of course, white space is used to separate tokens and keywords, but the amount is not important to the parser.

The short form of these kinds of statements does depend on statement context, however. First, note that the preceding code can also be written as

```
if (%a==1)% x=5;
```

This demonstrates that the braces in the earlier example are superfluous in this particular flavor of statement. However,

```
if (%a==1)
  %x=5;
```

is a valid rendition of the short form—*but the conditional code that you want executed must exist as a single statement that immediately follows the conditional test.* In this example, if the test is satisfied, `%x` is assigned the value 5. If the test is not satisfied, the ensuing assignment is not carried out.

However, using the same form

```
if (%a==1)
  %x=5; %b=6;
```

if the test is satisfied, `%x` is assigned the value 5 as before, and `%b` is assigned the value 6. But (and this is a big *but*) if the test is not satisfied, although the ensuing assignment statement is not carried out, the one after it *still is.* So with this last bit of code, `%b` always gets assigned the value 6.

By now you may be wondering why this digression—here's why: The `SoloScreen::onWake` method has the following statements that search for available mission files to use to populate its list:

```
for(%file = findFirstFile($Server::MissionFileSpec);
      %file !$= ""; %file = findNextFile($Server::MissionFileSpec))
   if (strStr(%file, "CVS/") == -1 && strStr(%file, "common/") == -1)
      SoloMissionList.addRow(%i++, getMissionDisplayName(%file) @ "\t" @ %file );
```

You might be tempted to misinterpret this code, even if you thoroughly understand programming in C or Torque Script. What we need to do is simplify the code to remove obfuscation introduced by the line context: We'll change all instances of `findFirstFile($Server::MissionFileSpec)` to `fFF()`, all instances of `findNextFile ($Server::MissionFileSpec))` to `fNF()`, and finally, all instances of `getMissionDisplayName (%file)` to `gMDN()`. Now the code will look like this (it won't compile, but we don't care about that):

```
for(%file = fFF();
         %file !$= ""; %file = fNF())
      if (strStr(%file, "CVS/") == -1 && strStr(%file, "common/") == -1)
         SoloMissionList.addRow(%i++, gMDN()@ "\t" @ %file );
```

If we tidy up the white space a bit, we get this:

```
for(%file = fFF(); %file !$= ""; %file = fNF())
      if (strStr(%file, "CVS/") == -1 && strStr(%file, "common/") == -1)
         SoloMissionList.addRow(%i++, gMDN()@ "\t" @ %file );
```

And hey, presto! The code structure reveals the algorithm quite nicely. The original line wrapping made the code hard to understand and made it look wrong when it actually wasn't. There are several lessons to be learned here:

1. Make sure your programming editor lets you display long lines of maybe 150 characters or more, just in case you have them.
2. Pay attention to your function and variable name lengths. Long descriptive names are extremely useful when you are trying to understand unfamiliar or long-forgotten code, but there are times when they can confuse more than explain.
3. Your own code may confuse you at some later point just as much as it might confuse someone else who needs to understand it (someone you've called in to fix bugs for you, for example).

What fix do I recommend for this? Shorter names? No, use braces and indenting and put the statements in the long form in order to remove any contextual ambiguity.

```
for(%file = findFirstFile($Server::MissionFileSpec);
      %file !$= ""; %file = findNextFile($Server::MissionFileSpec))
{
   if (strStr(%file, "CVS/") == -1 && strStr(%file, "common/") == -1)
   {
      SoloMissionList.addRow(%i++, getMissionDisplayName(%file) @ "\t" @ %file );
   } // end of if
 } // end of for
```

You can also add comments if they clarify what you are doing. Don't worry about insulting the intelligence of expert programmers by doing this. Any seasoned hand will greatly appreciate anything you do to make it quick and easy to understand what you are doing. *Especially* if they are doing code reviews for you!

Now, after that long-windedness, we can address the second issue about that code: What does it do?

The initial `findFirstFile` uses the variable to search the specified directory for the first instance of a matching file. If you actually *do* find a match, the path name is deposited in the `%file` variable, and you enter a loop. In each iteration of the loop, calls are made to `findNextFile`, which will find any new file in the sequence that matches the search criteria. If `findNextFile` does not find any more matching files, the `%file` variable is set to `NULL`, and the loop exits. In the loop we check the contents of the path name in `%file` for the existence of two potential invalid directory names: CVS (used for source code management, and *not* part of Torque) and common. If the file we found is not in either of those two directories, then we consider it to be valid and add it to our mission list using the `SoloMissionList.addRow` method.

The `findFirstFile-findNextFile` construct is a powerful one. It maintains an internal list of the files that it has found on your behalf. You just need to extract the path names as they appear.

Having said all that about such a small chunk of code, I should point out that this interface is a basic one. You might consider adding a few more capabilities, such as the sequence or random map selection option you'll find next in the `Host` interface.

The `getMissionDisplayName` is a bigger and more impressive-looking bit of work, but its function is fairly straightforward, albeit with a semimagical twist to it, so to speak. It opens up a file as directed and looks through it for the line that contains the statement `"%MissionInfoObject = "`. It then creates an actual `MissionInfoObject` using that statement and uses the `name` property of the object to obtain the name and return the name to the calling function. This is a pretty clever way to examine a file. Pretty sensible too, when you realize that mission files are simply Torque Script files with a different extension.

This bit of code presents to you a lot of possibilities about how you can use Torque Script. One that comes to mind is a reprogrammable AI robot, where you merely read in the new instructions at run time, with the instructions written in Torque Script. No need to create your own robot control language!

Host Interface Code

The `Host` interface code is similar to the `SoloPlay` code that you've just looked at. There is nothing remarkable about it that hasn't been already mentioned, except that you should

add some code to provide the player the ability to choose between playing maps in sequence (as exists now) or randomly.

You will want to have the Sequence and Random buttons that I've already provided in `HostScreen.gui` set a variable that your `onWake` code can examine. If the variable has one value, leave things as they are. If the variable has a different value, then have the `onWake` method choose a map randomly. One simple method to introduce the randomness is to select a random value between 0 and the number of available maps, and then to reject that many maps when the `findNextFile` function returns them. Then you would accept the next map returned.

Give it a try.

FindServer Interface Code

You saw the `FindServer` interface way back there in Figure 23.4. It lets you browse for servers with which you can connect. We've already looked at how this part of Torque works back in Chapter 5, 6, and 7, so I won't go into too much detail here. The functional code for the `FindServer` interface, extracted from ServerScreen.cs, is shown here for a brief discussion:

```
function ServerScreen::onWake()
{
   MasterJoinServer.SetActive(MasterServerList.rowCount() > 0);
}
function ServerScreen::Query(%this)
{
   QueryMasterServer(
      0,           // Query flags
      $Client::GameTypeQuery,       // gameTypes
      $Client::MissionTypeQuery,    // missionType
      0,           // minPlayers
      100,         // maxPlayers
      0,           // maxBots
      2,           // regionMask
      0,           // maxPing
      100,         // minCPU
      0            // filterFlags
      );
}
function ServerScreen::Cancel(%this)
{
   CancelServerQuery();
}
```

```
function ServerScreen::Join(%this)
{
   CancelServerQuery();
   %id = MasterServerList.GetSelectedId();
   %index = getField(MasterServerList.GetRowTextById(%id),6);
   if (SetServerInfo(%index)) {
      %conn = new GameConnection(ServerConnection);
      %conn.SetConnectArgs($pref::Player::Name);
      %conn.SetJoinPassword($Client::Password);
      %conn.Connect($ServerInfo::Address);
   }
}
function ServerScreen::Close(%this)
{
   cancelServerQuery();
   Canvas.SetContent(MenuScreen);
}
function ServerScreen::Update(%this)
{
   ServerQueryStatus.SetVisible(false);
   ServerServerList.Clear();
   %sc = getServerCount();
   for (%i = 0; %i < %sc; %i++) {
      setServerInfo(%i);
      ServerServerList.AddRow(%i,
         ($ServerInfo::Password? "Yes": "No") TAB
         $ServerInfo::Name TAB
         $ServerInfo::Ping TAB
         $ServerInfo::PlayerCount @ "/" @ $ServerInfo::MaxPlayers TAB
         $ServerInfo::Version TAB
         $ServerInfo::GameType TAB
         %i);  // ServerInfo index stored also
   }
   ServerServerList.Sort(0);
   ServerServerList.SetSelectedRow(0);
   ServerServerList.scrollVisible(0);

   ServerJoinServer.SetActive(ServerServerList.rowCount() > 0);
}
function onServerQueryStatus(%status, %msg, %value)
{
   if (!ServerQueryStatus.IsVisible())
```

```
      ServerQueryStatus.SetVisible(true);
   switch$ (%status) {
      case "start":
         ServerJoinServer.SetActive(false);
         ServerQueryServer.SetActive(false);
         ServerStatusText.SetText(%msg);
         ServerStatusBar.SetValue(0);
         ServerServerList.Clear();
      case "ping":
         ServerStatusText.SetText("Ping Servers");
         ServerStatusBar.SetValue(%value);
      case "query":
         ServerStatusText.SetText("Query Servers");
         ServerStatusBar.SetValue(%value);
      case "done":
         ServerQueryServer.SetActive(true);
         ServerQueryStatus.SetVisible(false);
         ServerScreen.update();
   }
}
```

Here the `OnWake` method makes the list active if there is anything already available from a previous incarnation to list. It's invoked as soon as the interface object is displayed on the screen.

When you click the Query Master button, the `Query` method is called, which sends a query packet to the master server, informing the master about what sort of servers are of interest. If the master server returns any information, it is deposited in the server information list, the `Update` method is invoked, and the list is created on the screen. This back-and-forth transaction is described in greater detail in Chapter 6.

The `onServerQueryStatus` method handles the various responses from the master server and deposits returned information, according to the changing states, into the various fields of the list.

ChatBox Interface Code

Open the file C:\koob\control\client\Initialize.cs and add the following lines to the function `InitializeClient`:

```
   Exec("./misc/ChatBox.cs");
   Exec("./misc/MessageBox.cs");
```

These exec statements load the new files that will provide our chat interface. You can copy them from C:\3DGPAi1\RESOURCES\CH23 and put them into the directories under the C:\koob\control\client\ directory in the subdirectories specified in the exec statements.

The ChatBox interface receives its text via a rather convoluted route. The message text originates at one of the clients and is sent to the server. The server receives the typed message and passes it to some common code that handles chat messages between the server and the client. Once the message arrives at the client common code, it is passed to the message handler called onChatMessage, which we provide in our client control code in our ChatBox.cs module. There is a parallel handler we are expected to supply in our client control code called onServerMessage, which is essentially the same as the one for the chat messages. These two functions look like this:

```
function onChatMessage(%message, %voice, %pitch)
{
   if (GetWordCount(%message)) {
      ChatBox.AddLine(%message);
   }
}
function onServerMessage(%message)
{
   if (GetWordCount(%message)) {
      ChatBox.AddLine(%message);
   }
}
```

Not much needed here—just add the new text to the ChatBox object using its AddLine method.

The AddLine method is where all the heavy lifting is done; it looks like this:

```
function ChatBox::addLine(%this,%text)
{
   %textHeight = %this.profile.fontSize;
   if (%textHeight <= 0)
      %textHeight = 12;
   %chatScrollHeight = getWord(%this.getGroup().getGroup().extent, 1);
   %chatPosition = getWord(%this.extent, 1) - %chatScrollHeight +
        getWord(%this.position, 1);
   %linesToScroll = mFloor((%chatPosition / %textHeight) + 0.5);
   if (%linesToScroll > 0)
      %origPosition = %this.position;
   while( !chatPageDown.isVisible() && MsgBoxMessageVector.getNumLines() &&
         (MsgBoxMessageVector.getNumLines() >= $pref::frameMessageLogSize))
```

```
   {
      %tag = MsgBoxMessageVector.getLineTag(0);
      if(%tag != 0)
         %tag.delete();
      MsgBoxMessageVector.popFrontLine();
   }
   MsgBoxMessageVector.pushBackLine(%text, $LastframeTarget);
   $LastframeTarget = 0;
   if (%linesToScroll > 0)
   {
      chatPageDown.setVisible(true);
      %this.position = %origPosition;
   }
   else
      chatPageDown.setVisible(false);
}
```

We start out by getting the font size from the profile. We need this in order to determine the height and width spacing requirements for scrolling and frame sizing.

Then we use `getGroup` to obtain the handle for the object group this control belongs to. And we use that handle to get the parent group's handle. Then we use that handle to get the `extent` property that tells us the height and width of the parent object. We take the second value in the extent—which is the height—by using `getWord` to get word number 1, which is actually the second word. (We perverted programmers usually count starting at 0 instead of 1—but not always!)

The object retains the current output position using the position parameter, and that is used to calculate where the next position will be and saved as `%chatPosition`. We then use the calculations to figure out `%linesToScroll`, which dictates the text scroll action and the scroll bar actions.

Next, we enter a loop that extracts text from the text buffer called `MsgBoxMessageVector` line by line and inserts the lines in the `ChatBox` control.

Finally, we adjust the visibility of the scroll down prompt based on whether or not our position causes text to be out of sight at the bottom of the display.

MessageBox Interface Code

The `MessageBox` interface accepts our input from the keyboard.

We need to add a message handler to the server to receive the typed messages when they are sent from the client. Because of the context, it makes more sense to do that here than in Chapter 22, even though we are dealing with client issues in this chapter.

Open the file C:\koob\control\server\server.cs and add the following function to the end of the file:

```
function serverCmdTypedMessage(%client, %text)
{
   if(strlen(%text) >= $Pref::Server::MaxChatLen)
      %text = getSubStr(%text, 0, $Pref::Server::MaxChatLen);
   ChatMessageAll(%client, '\c4%1: %2', %client.name, %text);
}
```

This handler grabs the incoming typed message, makes sure that it isn't too long (we may want to restrict chat messages in order to preserve bandwidth requirements), and then sends the message to the common code server function called `ChatMessageAll`. The `ChatMessageAll` function will distribute the message to all of the other clients logged in to our game.

Next, let's look at the code that manages this on behalf of the `MessageBox` interface:

```
function MessageBox::Open(%this)
{
   %offset = 6;
   if(%this.isVisible())
      return;
   %windowPos = "8 " @ ( getWord( outerChatFrame.position, 1 ) + getWord( outerChat-
Frame.extent, 1 ) + 1 );
   %windowExt = getWord( OuterChatFrame.extent, 0 ) @ " " @ getWord(
MessageBox_Frame.extent, 1 );
   %textExtent = getWord(MessageBox_Text.extent, 0);
   %ctrlExtent = getWord(MessageBox_Frame.extent, 0);
   Canvas.pushDialog(%this);
   MessageBox_Frame.position = %windowPos;
   MessageBox_Frame.extent = %windowExt;
   MessageBox_Edit.position = setWord(MessageBox_Edit.position, 0, %textExtent + %off-
set);
   MessageBox_Edit.extent = setWord(MessageBox_Edit.extent, 0, %ctrlExtent - %textExtent
- (2 * %offset));
   %this.setVisible(true);
   deactivateKeyboard();
   MessageBox_Edit.makeFirstResponder(true);
}
function MessageBox::Close(%this)
{
   if(!%this.isVisible())
      return;
```

```
   Canvas.popDialog(%this);
   %this.setVisible(false);
   if ( $enableDirectInput )
      activateKeyboard();
   MessageBox_Edit.setValue("");
}
function MessageBox::ToggleState(%this)
{
   if(%this.isVisible())
      %this.close();
   else
      %this.open();
}
function MessageBox_Edit::OnEscape(%this)
{
   MessageBox.close();
}
function MessageBox_Edit::Eval(%this)
{
   %text = trim(%this.getValue());
   if(%text !$= "")
      commandToServer('TypedMessage', %text);
   MessageBox.close();
}
function ToggleMessageBox(%make)
{
   if(%make)
      MessageBox.toggleState();
}
```

The `Open` method does some assignments of local variables based on the settings of properties of the `MainChatBox` object. This is so we can place the message box into a position relative to the chat display; in this case we are going to put it below and offset a little bit to the right.

Once we've done this, the code loads the `MessageBox` control into the Canvas using `Canvas.pushDialog(%this)`, where `%this` is the handle of the `MessageBox` control object, and positions it according to the values of the earlier saved local variables.

When we've completed the positioning of the control, then the code makes it visible.

Next, the code turns off keyboard input for the Canvas object and sets the `MessageBox_Edit` subobject responsible for handling key inputs. From this point on, all typing goes into the `MessageBox_Edit` subobject, until something changes that.

The `Close` method removes the control from the Canvas, makes the control invisible again, and restores keyboard input handling to the Canvas.

The `ToggleState` method merely opens or closes the message box in a toggle fashion. If the control is open, it closes it, and vice versa.

The `OnEscape` method closes the control. This method is defined as the `escapeCommand` property value in the interface definition in MessageBox.gui.

The `Eval` method obtains the entered text, trims empty spaces from the end, and sends the text to the server as the parameter for a `TypedMessage` message, which the server knows how to handle.

Finally, the `ToggleMessageBox` method is bound to the "t" key in our presets.cs file. When it receives a non-null value in `%make`, it changes the current `MessageBox` open state using the `ToggleState` method.

Game Cycling

The final feature we need to implement is the ability to cycle games when they are over—that is, when a player has reached either the score limit or the time limit.

First, add the following functions to the end of C:\koob\control\server\server.cs:

```
function cycleGame()
{
   if (!$Game::Cycling) {
      $Game::Cycling = true;
      $Game::Schedule = schedule(0, 0, "onCycleExec");
   }
}
function onCycleExec()
{
   endGame();
   $Game::Schedule = schedule($Game::EndGamePause * 1000, 0, "onCyclePauseEnd");
}

function onCyclePauseEnd()
{
   $Game::Cycling = false;
   %search = $Server::MissionFileSpec;
   for (%file = findFirstFile(%search); %file !$= "";
         %file = findNextFile(%search)) {
      if (%file $= $Server::MissionFile) {
         %file = findNextFile(%search);
```

```
         if (%file $= "")
           %file = findFirstFile(%search);
         break;
      }
   }
   loadMission(%file);
}
```

The first function, `cycleGame`, schedules the actual cycling code to occur at some later point. In this case we do it right away after making sure that we aren't actually already cycling.

The function `nCycleExec` actually ends the game. The `endGame` function just stops when it finishes, not doing anything else. Further action is scheduled to be taken by the `onCyclePauseEnd` function. This allows us to put up a victory screen or other messages and leave them up for an appropriate viewing time before continuing on to the next game.

In order to provoke the actual `cycleGame` function into being, we do two things. First, when the game is launched, we schedule its demise based on `$Game::Duration`. To do this, locate the function `StartGame` farther up in the server.cs file, and add these lines to the beginning:

```
if ($Game::Duration)
   $Game::Schedule = schedule($Game::Duration * 1000, 0, "CycleGame" );
```

This starts the game timer running. When it expires, it invokes the `cycleGame` function.

The other thing we need to do is add some code that checks to see if a player has hit the `$Game::MaxPoints` limit.

Locate the function `GameConnection::DoScore()` and add this code to the top of the function:

```
%client.score = (%client.lapsCompleted * $Game::Laps_Multiplier) +
                (%client.money * $Game::Money_Multiplier) +
                (%client.deaths * $Game::Deaths_Multiplier) +
                (%client.kills * $Game::Kills_Multiplier) ;
```

This code accumulates the various scoring values into a single overall score. Now add the following code to the end of the same `DoScore` function:

```
if (%client.score >= $Game::MaxPoints)
  cycleGame();
```

This causes the game cycling activity to happen if any one player hits the score limit. Game cycling entails ending the game, loading a new map, and dropping the players into the game in the new map.

Final Change

The final, very, very last piece of code we are going to change will allow us to remain in the program after we exit a game. Previously, when we exited a game using the Escape key, the program quit. This final change tidies that up for us. Open the file C:\koob\control\client\misc\presetkeys.cs and locate the function `DoExitGame()` and change it to match the following:

```
function DoExitGame()
{
   if ( $Server::ServerType $= "SinglePlayer" )
      MessageBoxYesNo( "Exit Mission", "Exit?", "disconnect();", "");
   else
      MessageBoxYesNo( "Disconnect", "Disconnect?", "disconnect();", "");
}
```

This function now checks to see if we are in single- or multiplayer mode. It does this to provide a customized exit prompt depending on which mode it is. In any event, the `disconnect` function is called to break the connection with the game server.

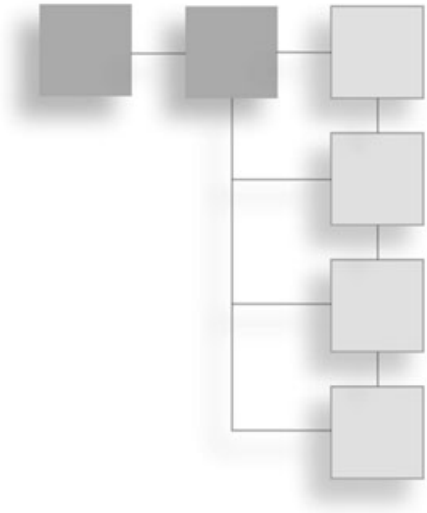
Moving Right Along

So there you have it. I hope your fingers aren't worn to the bone. You can see that there is a great deal of power available to those worn fingertips. I'm sure that as you've progressed through the preceding chapters, your head began to fill with all sorts of ideas about things you might want to do. In the next and final chapter of the book, I have a few things I want to say on that topic.

This page intentionally left blank

Team LRN

CHAPTER 24



The End Game

By now you've worn many hats, as programmer, 2D artist, 3D modeler, sound engineer, and level designer, to mention just the big ones. It should be fairly evident that each of these specialties has a great deal of depth, and it is hard to do justice to any one of them in a book like this.

However, it should also be apparent that you can make complex and feature-rich games without the need for million-dollar budgets. In this chapter we'll look at some of the things that didn't quite fit as topics in the earlier chapters.

A great deal of the work is done for us by the Torque Engine, but that's just where the process starts—the end is wherever you want it to be. There are other game engines out there, ranging from free to expensive, but the relationship between the end result and the price of the engine is not a linear one. The result is dependent on the amount of effort and inspiration *you* bring to the table. Making successful games is about transforming a great idea into a great game, and that operation can't be bought with mere money.

If you are going to put together a small team to develop a game using Torque, I would suggest you fill the artistic slots first—at least sign up a dedicated 3D modeler. You will also need one programmer to manage your script work. Finally, you need someone responsible for doing map layout, creating game rules, and managing the relationship between the models and the code. This makes a three-person team, which is probably as close to an ideal size as you're going to get for a small, low-budget development team. If you have the luxury of adding another team member, make sure you give him the sound-engineer responsibilities.

Team LRN

Testing

To properly test your game you are going to need to go back to your requirements and review them. For each specific requirement you have to decide what procedure *someone else* would need to perform to prove to you that *their* software fulfilled that requirement. Write the procedure down, and move on to the next requirement. Be hypercritical, a skeptic's skeptic.

Basics

There are many formalized testing methodologies, but the basic need when testing is to ensure at least two things about any feature:

1. Does the program feature (operation, appearance, behavior) work the way it should, when it should?
2. Does the feature make something else *not* work the way it should, when it should?

Take your list of test procedures and run through your software answering those two questions. It is certainly a lot tougher to answer the second question—sometimes you will see something not working, only to find out later that it was some *other* feature that was causing the problem that you witnessed.

You will end up with a list of problems and probably some ideas about how to fix them. Fill your list up first before running off to repair the bugs, and then sit back and examine the list of problems to try to identify problems that may share the same root cause. You can possibly save much effort and time by fixing the root cause. Otherwise you might end up with a series of individual fixes and hacks, each of which only addresses a single issue, and each of which exposes another issue.

Regression

The phenomenon *regression* is caused by bug fixes that introduce new bugs or sometimes expose hidden bugs. Some software engineers dispute the idea of referring to exposing hidden bugs as regression, but to me it's a difference without a distinction. The result is the same.

To deal with regression, we need to run our tests after every bug-fixing session. Ask the same questions and look for the answers. If you have the time and patience (neither of which is commonly overly abundant), you should run your regression test after each bug fix. In other words, don't do your entire list of bug-fixing programming all at once and then jump back to your regression test. If new bugs have been introduced, it may be hard to find them, because the new code can be quite extensive.

Team LRN

Play Testing

You will also want to enlist a bevy of play testers because there can be more wrong with the game than simple (or not-so-simple) bugs. You need to ensure that the game is fun to play, and you need to ensure that the things you can do in the game have the effect you want them to have. If your game features an Easter egg hunt, you want to make sure that the players can actually *find* the hidden items. At the same time, you will probably want to make sure that the items aren't too easy to find. Achieving the balance in game play is why you want to use play testers.

When you and members of your development team are testing the software, this is usually referred to as the *alpha* test phase. Alpha testing can be considered complete when the development team's own testing is no longer finding problems. This, however, doesn't mean that testing is finished! You will eventually need to use people who have not been involved in the creation of the game for testing. Once you start letting outsiders play-test your game, you are now in the *beta* test phase. If the game is fun (and it will be, right?) then you should not have much trouble interesting people in being beta testers. And this will introduce a problem (it's a problem you want to have, but still a problem), which is that many beta testers will be playing the game and not testing it. While it is good for them to be enjoying themselves, you need them to take notes and record problems, issues, and general feelings about the way the game is played. You need these notes in order to know what to fix and what to change or add.

Test Harnesses

You should also consider creating *test harnesses* to use in your testing. These are programs that are designed to provide the inputs that will cause the various features of the game to be exercised. The testing software should log its output to a file, automatically take screen shots, or record whatever else that is needed so that you can review the results.

For example, you could create a special version of the client that will automatically run and play as if it were a real player. Then you could launch dozens of these clients in order to simulate client loads on the server.

Hosted Servers

As you've seen with the example programs, with Torque there are three different execution modes:

- client only
- server only
- hosted server

Depending on your needs, you might want to create one monolithic program that will run in all three modes. This is certainly possible with Torque—in fact, the Torque demo as created by GarageGames supplies this capability by default.

However, you may want to create two or three different program distributions—one for each mode, or one for client only, the other being one or both of the server modes. There are some reasons for doing this, and probably the best is for server security. It depends on your business model (if you have one). If you are planning to provide all of the server-side hosting, then you might want a client-only version to be distributed to users. By not sending out the server code, you minimize the risk of unscrupulous players hacking the game to gain an advantage over other players in online play.

There are pitfalls to the multiple-version approach, the most noteworthy of which is the need to maintain two or more different versions of programs. That's a potential nightmare looking for a place to happen. Proceed with caution.

Having said all that, the distribution of multiplayer games that allow players to host other players while they all play in the same game is a common approach. Not only do many games offer it, but thousands of players use the capability.

Dedicated Servers

Some games, especially those that offer persistent role-playing style features, are hosted on dedicated servers only. The game's developer, or a service company that represents the developer, usually operates these servers. These games generally offer virtual environments where hundreds or thousands of players connect to the same world and interact in varying ways. This usually presents bandwidth and CPU costs well beyond the abilities of casual players and hobbyists looking for an evening's entertainment.

These sorts of "big iron" servers are often hosted on clustered servers at dedicated hosting service companies with battery-backup systems and racks and racks of computers.

This does not mean that you shouldn't offer your users the ability to run a dedicated server. There are many 16- or 32-player first-person shooter games on the market that have hordes of fans that run permanent 24/7 servers. Offering a dedicated server mode allows your users to run the servers on computers with less capability than they might otherwise use as their game-playing computer. That is to say, dedicated servers are an ideal way for users to utilize that two-year old computer sitting in the corner gathering dust!

FPS Game Ideas

You might be tempted to think that all the great first-person shooter ideas have already been done to death. I doubt that's true. There are a few ideas that have been tried and have not been terribly successful—that doesn't mean that they can never be successful. Maybe

with a bit of tweaking, you could make a successful version of a game that previously bombed. That's an important concept to keep in the back of your head.

One such example that immediately comes to mind is the Western—you know, the Wild, Wild West. There have been a ton of successful Hollywood Western movies. But there haven't been any equivalent games. That's an assignment for *someone* out there, and if it is ever going to be fulfilled, it will likely be an independent like you that does it.

One of the games I'd really like to see someone create in the FPS genre is a chess game played out with individual battles between pieces, where you can have each player able to engage in combat appropriate to the chess pieces as they are moved. There are game play issues that would need to be resolved, but that's something a clever designer would overcome. Here are some of the issues that would have to be tackled:

- Who decides the moves if the game is team based?
- Should each piece have different combat styles?
- Should the standard rules of chess play (movement rules, for example) prevail?
- Might you need to modify them slightly?
- Should you ever have an overhead board view?

If you broaden the scope a bit and don't focus on the *shooter* part of the FPS genre, the horizon starts to recede—first-person perspective play without the shooting has been barely touched.

Firefighting is one such topic that seems like it might be ripe for a game, especially team-based play. You could do forest fires, building fires, and so on. The biggest challenge would be the fire-propagation algorithms, such as the following:

- Exactly what conditions cause this item or that item to burst into flames?
- How does smoke move through a forest, a building, and so on, and how do you render that?
- How do you score the game?
- How realistic should the game be?

Other Genres

If you now shift a little to encompass third-person perspective play, the horizon opens up yet again. Almost any sport you can think of can be simulated from this point of view: bullfighting, surfing, rugby, and sailboarding, to name a few.

One that I would like to see is a mountain-biking game. I'd especially like to see one that accepted input from a stationary bicycle! Imagine being able to ride single-track trails at Moab while buried under three feet of snow in Ontario! That would be cool. In fact, I

think there is an untapped market here: hooking up the various machines in a gym to computers running games that people can play, using the exercise machines as the input devices. Exercise equipment manufacturers have put out some weak attempts at this, but there could be so much more—especially in the online multiplayer realm.

Instead of running on a treadmill hooked up to a computer with software that simulates running on a trail in Oregon, how about using the treadmill to provide motive input for your player as a rifleman in *World War II Online*? In fact, there is hardware that hasn't even been designed yet that could probably be used in this way.

Modifying and Extending Torque

If you sign up with GarageGames and buy a developer's license for their Torque Game Engine, you get all of the source code. Every single bit of it.

Stop and think about that for just a minute. Not only do you get the capabilities already described in this book—features you've been learning how to use to make your game—but you also get the access to the core engine code, with the right to change it as you like to make your game do *absolutely whatever you want it to do!*

Earlier I'd pointed out that Torque is not really designed for massively multiplayer games. With access to the source code, you could change that, adding the missing bits and modifying the existing bits to accommodate your needs.

How about huge, I mean gigantic, game worlds? You could do this by modifying the Terrain Manager code to accommodate *paging terrain*, where the game only loads the terrain in the immediate and viewable area of the player. You would probably need to make a special world creation tool for managing large worlds—a tool you would create with Torque.

If you go to the GarageGames Web site (http://www.garagegames.com) and click the Make Games button, you will find a user community that is large, active, and thriving. Several of the retail games made with Torque are included on the companion CD for this book. At the GarageGames forums you will see the developers of these games in continuous conversation with people designing and making their own games—every one of them an independent just like you.

As you browse around, make your way to the Resources postings, and you will find a whole slew of code modifications submitted by members of the community to enhance the core capabilities of the Torque Engine. In fact, you will find that a substantial number of the features that Torque now has that it *didn't* have when it was first released were added as submissions from the user-developer community.

In addition to extending the core capabilities, another reason for modifying the engine would be to move the more CPU-intensive parts of your game scripts into the core engine in order to improve the execution speed and sometimes even the memory footprint (how

Team LRN

much memory your game uses). To do these things you will have to learn how to program in C/C++ or at least obtain the services of a competent C/C++ programmer.

Go For It

As an independent game developer, you owe nothing to anyone except yourself and your family. That being the case, there is an important and sometimes underrecognized imperative for every independent game developer: *Have fun!*

Given that you've picked up this book, you probably already have some ideas rattling around inside your head, and you've been thinking about making them happen. Armed with the tools and knowledge from this book, you can afford to try them out without being afraid of wasting years of your life finding out that an idea didn't work.

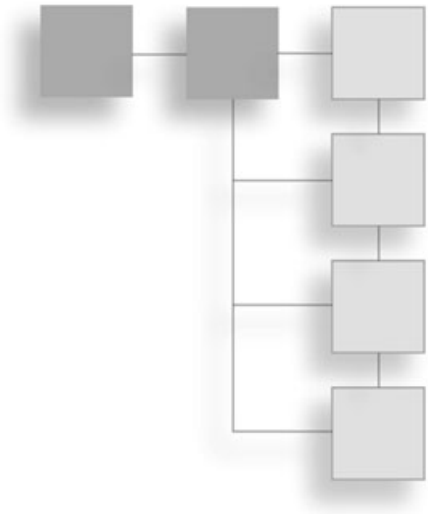
Now you can "waste" a few weeks finding out that an idea doesn't work, and then spend a few more weeks to refine it, some more weeks to tweak it, a few months to build on it, and finally, come up with something that might really fly.

Well, we are at the end of our journey. I hope you have enjoyed it as much as I have. I think the key thing to come away with is this: Believe in yourself.

This page intentionally left blank

Team LRN

Appendix A



The Torque Game Engine Reference

Torque Console Script Command Reference

activateDirectInput()

Parameters: *none*
Return: *nothing*
Description: Activates direct input device polling.
Usage:
```
activateDirectInput();
```

activateKeyboard()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Enables `DirectInput` polling of the keyboard.
Usage:
```
%result = activateKeyboard();
```

activatePackage(name)

Parameters: *name* String containing the name of the package.
Return: *nothing*
Description: Tells Torque to start using the package specified by name.
Usage:
```
activatePackage(Show);
```


Team LRN

AddCardProfile (vendor, renderer, safeMode, lockArray, subImage, fogTexture, noEnvColor, clipHigh, deleteContext, texCompress, interiorLock, skipFirstFog, only16, noArraysAlpha, profile)

Parameters:	*vendor*	Name of card vendor.
	renderer	Name of renderer.
	safeMode	`true` or `false`
	lockArray	`true` or `false`
	subImage	`true` or `false`
	fogTexture	`true` or `false`
	noEnvColor	`true` or `false`
	clipHigh	`true` or `false`
	deleteContext	`true` or `false`
	texCompress	`true` or `false`
	interiorLock	`true` or `false`
	skipFirstFog	`true` or `false`
	only16	`true` or `false`
	noArraysAlpha	`true` or `false`
	profile	Name of profile.
Return:	*nothing*	
Description:	Creates a profile of features of a video card for later reference.	
Usage:	`AddCardProfile(%vendor, %renderer, true, true, true, true, true, false, false, true, true, false, false, false,"")`	

addMaterialMapping(material, sound, color)

Parameters:	*material*	Name string to identify the material.
	sound	Name of sound profile to attach to material.
	color	Color specification to attach to material.
Return:	*nothing*	
Description:	Adds sound and dust color to specified material.	
Usage:	`addMaterialMapping("sand", "sound:0", "color:0.3 0.3 0.5 0.4 0.0" );`	

AddOSCardProfile(vendor,renderer,allowOpenGL,allowD3D,preferOpenGL)

Parameters:	*vendor*	Name of card vendor.
	Renderer	Name of renderer.
	AllowOpenGL	`true` or `false`
	allowD3D	`true` or `false`
	preferOpenGL	`true` or `false`
Return:	*nothing*	
Description:	Stores certain aspects of a video card for later usage.	
Usage:	`AddOSCardProfile(%vendor,%renderer,true,true,true);`	

AddTaggedString (string)

Parameters:	*string*	Normal string to be added.
Return:	*numeric*	The tag.
Description:	Adds a string to the tagged string list (`NetStringTable`).	
Usage:	`%tagname = AddTaggedString(%name);`	

AiConnect(id)

Parameters:	*id*	ID reference number (0 to 20) of the AI bot.
Return:	*numeric*	New object handle.
Description:	Creates a new uncontrolled AI connection. The AI is treated the same as a player.	
Usage:	`AiConnect(1);`	

alGetString(ALenum)

Parameters:	*string*	The enum string. Choices: "AL_VENDOR" "AL_VERSION" "AL_RENDERER" "AL_EXTENSIONS"
Return:	*string*	
Description:	Obtains the string specified.	
Usage:	`%vendor =  alGetString("AL_VENDOR");`	

alListener3f(ALenum, ["x y z"] | [x,y,z])

Parameters:	*ALenum*	The enum string. Choices: "AL_VELOCITY" "AL_POSITION" "AL_DIRECTION"
	"x y z"	The string contains a tuple indicating where to place the enumed property in 3D world space.
	x,y,z	(alternative) If "x y z" isn't used, then this is a tuple indicating where to place the audio object in 3D world space. *Note:* These are three numerics, not a string!
Return:	*nothing*	
Description:	Sets the *ALenum* to *value* for the listener (the player, who "hears" a sound).	
Usage:	`alListener3f("AL_GAIN_LINEAR", $pref::Audio::masterVolume );`	

AllowConnections(switch)

Parameters:	*switch*	1 (or `true`) = enable, 0 (or `false`) = disable.
Return:	*nothing*	
Description:	Enables and disables connections to the game server.	
Usage:	`AllowConnections(true);`	

alxCreateSource({ profile, [x,y,z] } | { description, filename, [x,y,z] })

Parameters:	*profile*	Descriptor string.
	x,y,z	If *profile* is used, this is a tuple indicating where to place the audio object. *Note:* These are three numerics, not a string!
	description	(alternative) If *profile* isn't used, then this is an audio object description string.
	filename	If *description* is used, then this string specifies the audio file to use for the sound.
	x,y,z	If *description* is used, this is a tuple indicating where to place the audio object. *Note:* These are three numerics, not a string!
Return:	*numeric*	Handle to audio object.
Description:	Loads an audio source file into memory, and initializes it for use.	
Usage:	`$handle =alxCreateSource("Audio0","~/data/sounds/test.wav");`	

alxGetChannelVolume(channel)

Parameters:	*channel*	Channel ID number.
Return:	*numeric*	
Description:	Queries the volume of *channel*.	
Usage:	`%vol = alxGetChannelVolume(%channel);`	

alxGetListener3f(ALenum)

Parameters:	*ALenum*	The enum string. Choices: "AL_VELOCITY" "AL_POSITION" "AL_DIRECTION"
Return:	*numeric*	
Description:	Queries the value of the *ALenum*.	
Usage:	`%direction = alxGetListener3f("AL_DIRECTION");`	

alxGetListenerf(ALenum)

Parameters:	*ALenum*	The enum string. Choices: "AL_GAIN" "AL_GAIN_LINEAR"
Return:	*numeric*	
Description:	Queries the value of the *ALenum*.	
Usage:	`%gain = alxGetListenerf("AL_GAIN");`	

alxGetListeneri(ALenum)

Parameters:	*ALenum*	The enum string. Choices: "AL_CONE_INNER_ANGLE" "AL_CONE_OUTER_ANGLE" "AL_LOOPING" "AL_STREAMING" "AL_BUFFER"
Return:	*numeric*	
Description:	Queries the value of the *ALenum*.	
Usage:	`%looping = alxGetListeneri("AL_LOOPING");`	

alxGetSource3f(handle, ALenum)

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices:
		"AL_VELOCITY"
		"AL_POSITION"
		"AL_DIRECTION"
Return:	*string*	"x y z".
Description:	Obtains the value of *ALenum* for the specified *handle*.	
Usage:	`%pos = alxGetSource3f(%handle[%sender], "AL_POSITION");`	

alxGetSourcef(handle, ALenum)

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices:
		"AL_PITCH"
		"AL_REFERENCE_DISTANCE"
		"AL_MAX_DISTANCE"
		"AL_CONE_OUTER_GAIN"
		"AL_GAIN"
		"AL_GAIN_LINEAR"
Return:	*numeric*	
Description:	Obtains the value of *ALenum* for the specified *handle*.	
Usage:	`%gain = alxGetSourcef(%handle[%sender], "AL_GAIN");`	

alxGetSourcei(handle, ALenum)

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices:
		"AL_CONE_INNER_ANGLE"
		"AL_CONE_OUTER_ANGLE"
		"AL_LOOPING"
		"AL_STREAMING"
		"AL_BUFFER"
Return:	*numeric*	
Description:	Obtains the value of *ALenum* for the specified *handle*.	
Usage:	`%pitch = alxGetSourcei((%handle[%sender], "AL_PITCH");`	

Team LRN

alxIsPlaying(handle)

Parameters:	*handle*	Handle to audio object.
Return:	*numeric*	1 = `true`, 0 = `false`.
Description:	Queries if a *handle* is currently playing.	
Usage:	`%isPlaying = alxIsPlaying(%handle);`	

alxListener(ALenum,value)

Parameters:	*ALenum*	The enum string. Choices: "AL_GAIN" "AL_GAIN_LINEAR"
	value	Numeric gain value.
Return:	*nothing*	
Description:	Sets the *ALenum* to *value* for the Listener (the player, who "hears" a sound).	
Usage:	`alxListener("AL_GAIN_LINEAR", %vol );`	

alxPlay([handle] | [profile [, x,y,z]])

Parameters:	*handle*	Handle to audio object.
	profile	(alternative) Descriptor string.
	x,y,z	If *profile* is used, this is a tuple indicating where to place the audio object. *Note:* These are three numerics, not a string! (optional)
Return:	*numeric*	Returns object handle if profile is used.
Description:	Begins audio playback with audio object specified by *handle*. Alternatively, if *profile* is used, this function creates an object, begins playback at optional *x,y,z* coordinates, and then returns a handle to the created object.	

Usage:	`%handle0 = alxCreateSource("Audio0", "~/data/sounds/test.wav");`
	`alxPlay(%handle0);`
	`%handle1 = alxPlay("Audio1", "100, 100, 10");`	

alxSetChannelVolume(channel, volume)

Parameters:	*channel*	Channel ID number.
	volume	Volume value.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Sets the *channel* to *volume*.	
Usage:	`%result = alxSetChannelVolume(%channel, %volume);`	

alxSource3f(handle,ALenum, ["x y z"] | [x,y,z])

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices: "AL_VELOCITY" "AL_POSITION" "AL_DIRECTION"
	"x y z"	String containing a tuple indicating where to place the enumed property in 3D world space.
	x,y,z	(alternative) If "xyz" isn't used, then this is a tuple indicating where to place the audio object in 3D world space. *Note:* These are three numerics, not a string!
Return:	*nothing*	
Description:	Sets *ALenum* for the specified *handle* to 3D.	
Usage:	`alxSource3f(%handle[%sender], "AL_POSITION", "100 100 20");`	

alxSourcef(handle, ALenum, value)

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices: "AL_PITCH" "AL_REFERENCE_DISTANCE" "AL_MAX_DISTANCE" "AL_CONE_OUTER_GAIN" "AL_GAIN" "AL_GAIN_LINEAR"
	value	Numeric (floating point) value to set *ALenum* to.
Return:	*nothing*	
Description:	Sets *ALenum* for the specified *handle* to the floating point value.	
Usage:	`alxSourcef(%handle[%sender], "AL_GAIN", %gain);`	

alxSourcei(handle, ALenum, value)

Parameters:	*handle*	Handle to audio object.
	ALenum	The enum string. Choices: "AL_CONE_INNER_ANGLE" "AL_CONE_OUTER_ANGLE" "AL_LOOPING" "AL_STREAMING" "AL_BUFFER"
	value	Numeric value to set *ALenum* to.
Return:	*nothing*	
Description:	Sets *ALenum* for the specified *handle* to the floating point *value*.	
Usage:	`alxSourcei(%handle[%sender], "AL_LOOPING", true);`	

alxStop(handle)

Parameters:	*handle*	Handle to audio object.
Return:	nothing	
Description:	Stops the playback from *handle*.	
Usage:	`alxStop(%handle[%sender]);`	

alxStopAll()

Parameters:	*none*
Return:	*nothing*
Description:	Stops the playback on all registered channels.
Usage:	`alxStopAll();`

Backtrace()

Parameters:	*none*
Return:	*nothing*
Description:	Enables echo of script call stack to console.
Usage:	`Backtrace();`

Team LRN

BuildTaggedString(string, format)

Parameters:	*string*	Normal string to be added.
	format	Format specifying string.
Return:	*string*	
Description:	Builds and adds a tagged *string* using string with specified *format*.	
Usage:	`%tagstring = BuildTaggedString(%name, %format);`	

CalcExplosionCoverage(location, handle, mask)

Parameters:	*location*	Where the target object is.
	handle	Target object.
	mask	Object type mask of objects that may block the explosion.
Return:	*numeric*	1 = affected, 0 = unaffected.
Description:	Determines if an object at a location was affected by an explosion. Listed object types will be taken into consideration in the calculation, if they would block the explosion force.	
Usage:	`%coverage = CalcExplosionCoverage(%location, %targetObject, $TypeMasks::InteriorObjectType | $TypeMasks::TerrainObjectType | $TypeMasks::VehicleObjectType);`	

Call(function [,args ...])

Parameters:	*function*	String containing name of function.
	args	Zero or more arguments as needed by *function*.
Return:	*string*	*Function's* return value embedded in a string.
Description:	Executes the function with the name *function* with supplied arguments, *args*.	
Usage:	`%result = Call(%func, %arg1, %arg2);`	

Cancel(id)

Parameters:	*id*	ID number of supposed event.
Return:	*nothing*	
Description:	Cancels the event specified by *id*.	
Usage:	`cancel($eventid);`	

CancelServerQuery()

Parameters: *none*
Return: *nothing*
Description: Cancels the current query and drops anything outstanding in the ping list.
Usage:
```
CancelServerQuery();
```

ClearTextureHolds()

Parameters: *none*
Return: *numeric* Amount of memory released.
Description: Releases any textures not being used, and frees the memory.
Usage:
```
%clearedMem=ClearTextureHolds();
```

CollapseEscape(text)

Parameters: *text* String.
Return: *string* The resultant string.
Description: Removes escaped characters in text. For example, `\\n` becomes `\n`.
Usage:
```
%coltext = CollapseEscape(%text);
```

CommandToClient(client, function [,arg1,...argn])

Parameters: *client* Handle of target client.
function Function on the server to be executed.
arg1,...argn Arguments for the function.
Return: *nothing*
Description: Tells client to execute the command specified by *function*, and passes it the arguments. On the client, the function is declared in the following format:
```
function ClientCmdfunction(arg1,...argn) { … }
```
The identifier `ClientCmd` is prepended to the function name.
Usage:
```
CommandToClient( %client, 'SyncClock', %time);
```

CommandToServer(function [,arg1,...argn])

Parameters:	*function*	Function on the server to be executed.
	arg1,...argn	Arguments for the function.
Return:	*nothing*	
Description:	Tells the server to execute the command specified by *function*, and passes it the arguments. On the server, the function is declared in the following format: `function ServerCmdfunction(%client, arg1,...argn) { … }` The identifier `ServerCmd` is prepended to the function name. The first parameter is always the handle of the client that sent the command, and the actual command arguments follow.	
Usage:	`CommandToServer('ToggleCamera');`	

Compile(fileName)

Parameters:	*filename*	String containing file name.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Compiles the source script file *filename*.	
Usage:	`%result = Compile("/common/default.cs");`	

ContainerBoxEmpty(mask, loc, rad [,yrad, zrad])

Parameters:	*mask*	Object type mask.
	loc	Coordinate tuple.
	rad	Radius distance (or X-axis distance).
	yrad	Optional distance in Y-axis.
	zrad	Optional distance in Z-axis.
Return:	*numeric*	
Description:	Returns `true` if any objects of given types exist in a sphere of the specified extent *rad* and `false` otherwise. If *yrad* is specified, then *rad* is the X-axis extent, and *yrad* is the Y-axis extent. If *zrad* is specified, it becomes the Z-axis extent.	
Usage:	`%isAny = ContainerBoxEmpty(ItemObjectType,"10.0 10.0 100.0", 100);`	

ContainerFindFirst(type, point, x, y, z)

Parameters:	*type*	The type mask of objects to find.
	point	Location of container.
	x,y,z	Numeric bounds of container specified. Not a string.
Return:	*numeric*	Handle of object found
Description:	Finds objects of *type* within the box specified with *x,y,z* at the given *point*. Returns the handle of the first object found.	
Usage:	`%objectHandle = ContainerFindFirst(type, point, x, y, z);`	

ContainerFindNext()

Parameters:	*none*	
Return:	*numeric*	Handle of object found.
Description:	Find the next object in the container specified by immediately previous call to `containerFindFirst`, and get its handle.	
Usage:	`%objectHandle = ContainerFindNext();`	

ContainerRayCast (start, end, mask, [exclude])

Parameters:	*start*	Starting coordinate tuple.
	end	Ending coordinate tuple.
	mask	Object type mask.
	exclude	List of handles.
Return:	*string*	Hit list.
Description:	Finds a list of objects of type *mask* between the two cords supplied. A list of object handles can be included in the *exempt* parameter that will not be returned in the hit list.	
Usage:	`%tgt = ContainerRayCast (%cameraPoint, %rangeEnd, ItemObjectType);`	

ContainerSearchCurrDist()

Parameters:	*none*	
Return:	*numeric*	
Description:	Gets the current container search distance.	
Usage:	`%dist = ContainerSearchCurrDist();`	

ContainerSearchCurrRadiusDist()

Parameters: *none*
Return: *numeric*
Description: Gets the current container search radius distance.
Usage:
```
%rad = ContainerSearchCurrRadiusDist();
```

ContainerSearchNext()

Parameters: *none*
Return: *numeric*
Description: Gets the next object in a container search.
Usage:
```
%nc = ContainerSearchNext();
```

CreateCanvas(title)

Parameters: *title* String containing title of the window.
Return: *numeric* 1 = success, 0 = fail.
Description: Creates a graphics canvas in a window.
Usage:
```
%result = CreateCanvas("My Game");
```

DbgSetParameters (port, pw)

Parameters: *port* Connection port.
pw Password.
Return: *nothing*
Description: Initializes telnet debug connection request parameters.
Usage:
```
DbgSetParameters(1130, "games");
```

DeactivateDirectInput()

Parameters: *none*
Return: *nothing*
Description: Disables DirectInput device polling (mouse, keyboard, joystick).
Usage:
```
DeactivateDirectInput();
```

Team LRN

DeactivateKeyboard()

Parameters: *none*
Return: *nothing*
Description: Disables DirectInput polling of the keyboard.
Usage:
```
DeactivateKeyboard();
```

DeactivatePackage(name)

Parameters: *name* String containing the name of the package.
Return: *nothing*
Description: Tells Torque to stop using the package specified by *name*.
Usage:
```
DeactivatePackage(Show);
```

Debug()

Parameters: *none*
Return: *nothing*
Description: Enables debug mode.
Usage:
```
Debug();
```

Debug_debugbreak()

Parameters: *none*
Return: *nothing*
Description: *Windows:* Displays a crash dialog box.
Linux: Causes a *segfault*.
Usage:
```
Debug_debugbreak();
```

Debug_testx86unixmutex()

Parameters: *none*
Return: *nothing*
Description: *Linux only:* Check if OS can create a mutex.
Usage:
```
Debug_testx86unixmutex();
```

DecreaseFSAA()

Parameters: *none*
Return: *nothing*
Description: Invokes `DecreaseNPatch`.
Usage:
```
DecreaseFSAA ()
```

DecreaseNPatch()

Parameters: *none*
Return: *nothing*
Description: Decrements `Npatch` level by 1.
Usage:
```
DecreaseNPatch():
```

DeleteDataBlocks()

Parameters: *none*
Return: *nothing*
Description: Unloads and removes all registered datablocks from the game.
Usage:
```
DeleteDataBlocks();
```

DeleteVariables(wildcard)

Parameters: *wildcard* Match string to specify variables.
Return: *nothing*
Description: Deletes global variables specified by *wildcard*. The wildcard string supports `"*"` to match any number of any characters and `"?"` to match any single character.
Usage:
```
DeleteVariables("*");
```

Detag(tstring)

Parameters: *tstring* Tagged string.
Return: *string* String value for the tagged string.
Description: Detags a tagged string.
Usage:
```
%name = Detag(%test.name);
```

DisableMouse()

Parameters: *none*
Return: *nothing*
Description: Disables `DirectInput` polling of the mouse device.
Usage:
```
DisableMouse();
```

DnetSetLogging(switch)

Parameters: *switch* 1 (or `true`) = enable, 0 (or `false`) = disable.
Return: *nothing*
Description: Enables network packet logging to the console.
Usage:
```
DnetSetLogging(1);
```

DumpConsoleClasses()

Parameters: *none*
Return: *nothing*
Description: Dumps all registered console classes to the console.
Usage:
```
DumpConsoleClasses();
```

DumpMemSnapshot(filename)

Parameters: *filename* String containing file name.
Return: *nothing*
Description: Dumps memory statistics to the file.
Usage:
```
DumpMemSnapshot("dump.txt");
```

DumpNetStringTable()

Parameters: *none*
Return: *nothing*
Description: Dumps the `NetStringTable` stats to the console.
Usage:
```
DumpNetStringTable();
```

DumpResourceStats();

Parameters: *none*
Return: *nothing*
Description: Dumps texture information to the console in the following format: `path`, `resource`, `lockCount`.
Usage:
```
DumpResourceStats();
```

DumpTextureStats()

Parameters: *none*
Return: *nothing*
Description: Dumps texture information to the console in the following format: `type`, `refCount`, `holding` (yes or no), `textureSpace`, `texFileName`.
Usage:
```
DumpTextureStats();
```

Echo(text)

Parameters: *text* String.
Return: *nothing*
Description: Prints *text* to the console with standard font. Text can be formatted according to the string rules.
Usage:
```
Echo( "Hello World");
```

EchoInputState()

Parameters: *none*
Return: *nothing*
Description: Displays the current state of DirectInput (mouse, keyboard, and joystick).
Usage:
```
EchoInputState();
```

EnableMouse()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Enables `DirectInput` polling of the mouse device.
Usage:
```
%result = EnableMouse();
```

EnableWinConsole(switch)

Parameters:	*switch*	1 enables, 0 disables.
Return:	*nothing*	
Description:	Displays the console window.	
Usage:	`EnableWinConsole(true);`	

Error(text)

Parameters:	*text*	String.
Return:	*nothing*	
Description:	Prints *text* to the console with red font. Text can be formatted according to the string rules.	
Usage:	`Error("I'm sorry, Dave, I'm afraid I can't do that.");`	

Exec(fileName [, nocalls])

Parameters:	filename	String containing file name.
	nocalls	When set to `true`, prevents functions from being called.
Return:	*string*	
Description:	Compiles, executes functions, assigns variables, and loads packages and data blocks within the file *filename*. If *nocalls* is set to `true`, functions are not executed, but the other operations still take place.	
Usage:	`%result = Exec("/common/default.cs");`	

ExpandEscape(text)

Parameters:	*text*	String.
Return:	*string*	The resultant string.
Description:	Escapes all of the escape characters in text. For example, `\n` becomes `\\n`. In this case the `\n` would be printed to the console instead of the new line it would otherwise cause.	
Usage:	`%extext = ExpandEscape(%text);`	

ExpandFilename(filename)

Parameters:	*filename*	String containing file name.
Return:	*string*	
Description:	Obtains the actual OS-specific absolute path of *filename*.	
Usage:	`%fullmissionpath = ExpandFilename("~/data/missions/test.mis");`	

Team LRN

Export(searchString [, fileName [,append]])

Parameters:	*search*	Prefix of variables to search for.
	filename	String containing file name.
	append	Indicates whether to append to file or overwrite.
Return:	*nothing*	
Description:	Saves the values of variables starting with *search* to the file named *filename*. When *append* is set to `true`, the file is appended; when set to `false`, the file is overwritten. The search string supports `"*"` to match any number of any characters and `"?"` to match any single character.	
Usage:	`%result = Export("$Pref::Game::*", "./game/prefs.cs", False);`	

FileBase(filename)

Parameters:	*filename*	String containing full file name.
Return:	*string*	String containing the base name.
Description:	Gets the base name of the file specified by *filename*.	
Usage:	`%base = FileBase("/common/server/script.cs");`	

FileExt(filename)

Parameters:	*filename*	String containing full file name.
Return:	*string*	String containing extension.
Description:	Gets the extension of the file specified by *filename*.	
Usage:	`%name = FileExt("script.cs");`	

FileName(filename)

Parameters:	*filename*	String containing full file name.
Return:	*string*	String containing the name.
Description:	Gets the name of the file specified by *filename*.	
Usage:	`%name = FileName("scripts.cs");`	

FilePath(filename)

Parameters:	*filename*	String containing full file name.
Return:	*string*	String containing the path.
Description:	Gets the path of the file specified by *filename*.	
Usage:	`%path = FilePath("/common/server/script.cs");`	

Team LRN

FindFirstFile (pattern)

Parameters:	*pattern*	String pattern.
Return:	*string*	The file's name.
Description:	Finds the name of the first file in the Torque Script file name buffer matching the given *pattern*. Supports `"*"` to match any number of any characters and `"?"` to match any single character.	
Usage:	`%result = FindFirstFile("/common/*.cs");`	

FindNextFile (pattern)

Parameters:	*pattern*	String pattern.
Return:	*string*	The file's name.
Description:	Finds the name of the next file in the Torque Script filename buffer matching the search by immediately previous call to `FindFirstFile`. Supports '*' to match any number of any characters and '?' to match any single character.	
Usage:	`%result = FindNextFile("/common/*.cs");`	

FirstWord(text)

Parameters:	*text*	String with space-delimited words.
Return:	*string*	The resultant string.
Description:	Gets the first word-string within *text*.	
Usage:	`%tgt = FirstWord(%text);`	

FlushTextureCache()

Parameters:	*none*
Return:	*nothing*
Description:	Deletes cached textures from memory.
Usage:	`FlushTextureCache();`

FreeMemoryDump()

Parameters:	*none*
Return:	*nothing*
Description:	Prints free memory statistics.
Usage:	`FreeMemoryDump();`

Team LRN

GetBoxCenter(box)

Parameters: *box* String containing two 3D tuples defining the box.
Return: *string*
Description: Computes the center of a box.
Usage:
```
%c = GetBoxCenter("10,10,10,50,50,50");
```

GetBuildString()

Parameters: *none*
Return: *string*
Description: Obtains the BUILD type (Release or Debug) of the current build.
Usage:
```
%bs = GetBuildString();
```

GetCompileTimeString()

Parameters: *none*
Return: *string*
Description: Obtains the compile time and date of the current build.
Usage:
```
%ct = GetCompileTimeString();
```

GetControlObjectAltitude()

Parameters: *none*
Return: *numeric*
Description: Obtains the altitude of the player object.
Usage:
```
%altitude = %player.GetControlObjectAltitude();
```

GetControlObjectSpeed()

Parameters: *none*
Return: *numeric*
Description: Obtains the speed of the player object.
Usage: %speed = `%player.GetControlObjectSpeed();`

GetDesktopResolution()

Parameters: *none*
Return: *string*
Description: Reports the current desktop resolution.
Usage:
```
%res = GetDesktopResolution():
```

GetDisplayDeviceList()

Parameters: *none*
Return: *string*
Description: Obtains the device name for each display device.
Usage:
```
%name = GetDisplayDeviceList();
```

GetField(text, index)

Parameters: *text* String with field-delimited words.
index Field-based offset into the text string.
Return: *string* Contains the found field-string.
Description: Gets the field-string at *index* within *text*. In the usage example that follows, if `%text` equaled "Of Mice and Men", then `%word` would be set to "and" when the function returned.
Usage:
```
%field = GetField(%text, 0);
```

GetFieldCount (text)

Parameters: *text* String with field-delimited words.
Return: *numeric*
Description: Gets the number of field-strings within *text*.
Usage:
```
%count = GetFieldCount( %text );
```

GetFields(text, first [, last])

Parameters:	*text*	String with space-delimited fields.
	first	Field-based offset into the text string specifying the first field to extract.
	last	Field-based offset into the text string specifying the last field to extract.
Return:	*string*	Contains the found fields.
Description:	Gets one or more field-strings at *index* within *text*. If *count* is specified, gets *count* number of field-strings.	
Usage:	`%position = GetFields(%obj.getTransform(), 0, 2);`	

GetFileCount (pattern)

Parameters:	*pattern*	String pattern.
Return:	*numeric*	
Description:	Gets the number of files in the Torque Script file name buffer that match *pattern*.	
Usage:	`%count = GetFileCount("/common/server/*.cs");`	

GetFileCRC(filename)

Parameters:	*filename*	String containing full file name.
Return:	*numeric*	The *Cyclic Redundancy Check* (CRC) value.
Description:	Gets the CRC value of the file specified by *filename*.	
Usage:	`%crc = GetFileCRC("/common/server/script.cs");`	

GetJoystickAxes(instance)

Parameters:	*instance*	The joystick object.
Return:	*string*	
Description:	Obtains the current axes of the joystick pointed to by *instance*.	
Usage:	`%joyAxes = GetJoystickAxes( 3 );`	

GetMaxFrameAllocation()

Parameters:	*none*
Return:	*numeric*
Description:	Gets the Max Frame Allocation unit.
Usage:	`%maxFrameAlloc = GetMaxFrameAllocation();`

Team LRN

GetModPaths()

Parameters: *none*
Return: *string*
Description: Gets the current Mod path.
Usage:
```
$mp = GetModPaths();
```

GetRandom([[max]|[min,max]])

Parameters: *max* High limit. (optional)
min Low limit. (optional)
Return: *numeric* Ranged from 0 to 1, exclusive, if no parameters given, otherwise, see description.
Description: Computes a pseudo-random number. If *min* is not included, then 0 is the minimum. If *max* is not included, then 4,294,967,296 (highest 32-bit number minus 1) is the maximum.
Usage:
```
%random = GetRandom(1,99);
```

GetRandomSeed()

Parameters: *none*
Return: *numeric*
Description: Obtains the current random seed.
Usage:
```
%seed = GetRandomSeed();
```

GetRealTime()

Parameters: *none*
Return: *numeric*
Description: Gets the real time (in milliseconds) since this computer started.
Usage:
```
%rt = GetRealTime();
```

GetRecord (text, index)

Parameters: *text* String with new line-delimited records.
index Record-based offset into the text string.
Return: *string* Contains the found record-string.
Description: Gets the record-string at index within text. In the usage example that follows, if `%text` equaled "Of Mice and Men\nGrapes of Wrath\nCannery Row", then `%record` would be set to "Grapes of Wrath" when the function returned.
Usage:
```
%record = GetRecord(%text, 1);
```

Team LRN

GetRecordCount (text)

Parameters:	*text*	String with new line-delimited records.
Return:	*numeric*	
Description:	Get the number of record-strings within *text*.	
Usage:	`%count = GetRecordCount( %text );`	

GetRecords (text, first [, last])

Parameters:	*text*	String with new line-delimited records.
	first	Record-based offset into the text string specifying the first record to extract.
	last	Record-based offset into the text string specifying the last record to extract.
Return:	*string*	Contains the found records.
Description:	Gets one or more record-strings at *index* within *text*. If *count* is specified, gets *count* number of record-strings.	
Usage:	`%books = GetRecords(%obj.getTransform(), 0, 2);`	

GetResolution()

Parameters:	*none*
Return:	*string*
Description:	Obtains the current screen resolution.
Usage:	`%res = GetResolution():`

GetResolutionList(devicename)

Parameters:	devicename Name of the device to query.
Return:	*string*
Description:	Obtains all available resolutions for the specified device.
Usage:	`%rl = GetResolutionList(%device);`

GetServerCount()

Parameters:	*none*
Return:	*numeric*
Description:	Gets the number of available servers from the master server.
Usage:	`%sc = GetServerCount();`

Team LRN

GetSimTime()

Parameters: *none*
Return: *numeric*
Description: Gets the current game time.
Usage:
```
%st =  GetSimTime();
```

GetSubStr(str, loc, count)

Parameters:	*str*	String to be processed.
	loc	Offset into *str* to get start of substring from.
	count	Number of characters to get.
Return:	*string*	The processed resultant string.

Description: Gets the substring of *string* that begins at *loc*, continuing for *count* characters or to the end of the string, whichever comes first.
Usage:
```
%sub = GetSubStr(%text, 5, 99);
```

GetTag(tstring)

Parameters:	*tstring*	Tagged string.
Return:	*string*	

Description: Gets the tag for the tagged string *tstring*.
Usage:
```
%tag = GetTag(%variable);
```

GetTaggedString(tag)

Parameters:	*tag*	Numeric tag of string to be removed.
Return:	*string*	

Description: Gets the string associated with *tag*.
Usage:
```
%name = GetTaggedString( %tagname );
```

GetTerrainHeight(pos)

Parameters:	*pos*	2D coordinate.
Return:	*numeric*	

Description: Gets the terrain height at the specified position.
Usage:
```
%height = GetTerrainHeight(%pos);
```

GetVersionNumber()

Parameters: *none*
Return: *numeric*
Description: Obtains the hard-coded engine version number of the current build.
Usage:
```
%vn = GetVersionNumber();
```

GetVersionString()

Parameters: *none*
Return: *string*
Description: Obtains the hard-coded engine version string of the current build.
Usage:
```
%vs = GetVersionString ();
```

GetVideoDriverInfo()

Parameters: *none*
Return: *string*
Description: Gets device driver information.
Usage:
```
%info = GetVideoDriverInfo();
```

GetWord(text, index)

Parameters: *text* String with space-delimited words.
index Word-based offset into the text string.
Return: *string* Contains the found word-string.
Description: Gets the word-string at *index* within *text*. In the usage example that follows, if `%text` equaled "Of Mice and Men", then `%word` would be set to "and" when the function returned.
Usage:
```
%word = GetWord(%text, 2);
```

GetWordCount(text)

Parameters: *text* String with space-delimited words.
Return: *numeric*
Description: Gets the number of word-strings within *text*.
Usage:
```
%count = GetWordCount( %text );
```

GetWords(text, first [, last])

Parameters:	*text*	String with space-delimited words.
	first	Word-based offset into the text string specifying the first word to extract.
	last	Word-based offset into the text string specifying the last word to extract.
Return:	*string*	Contains the found words.
Description:	Gets one or more word-strings at *index* within *text*. If *count* is specified, gets *count* number of word-strings.	
Usage:	`%position = GetWords(%obj.getTransform(), 0, 2);`	

GLEnableLogging(switch)

Parameters:	*switch*	1 enables, 0 disables.
Return:	*nothing*	
Description:	Enables OpenGL logging to gl_log.txt.	
Usage:	`GLEnableLogging(true);`	

GLEnableMetrics(switch)

Parameters:	*switch*	1 enables, 0 disables.
Return:	*nothing*	
Description:	Tracks metrics data for OpenGL features.	
Usage:	`GLEnableMetrics(1);`	

GLEnableOutline(switch)

Parameters:	*switch*	1 enables, 0 disables.
Return:	*nothing*	
Description:	Enables OpenGL wire-frame mode.	
Usage:	`GLEnableOutline(true);`	

GotoWebPage(address)

Parameters:	*address*	URL of Web page.
Return:	*nothing*	
Description:	Opens default browser with specified address.	
Usage:	`GotoWebPage("http://www.tubettiworld.com/");`	

Team LRN

IncreaseFSAA()

Parameters: *none*
Return: *nothing*
Description: Invokes `IncreaseNPatch`.
Usage: `IncreaseFSAA ()`

IncreaseNPatch()

Parameters: *none*
Return: *nothing*
Description: Increments `Npatch` level by 1.
Usage: `IncreaseNPatch();`

InitContainerRadiusSearch (loc, radius, mask)

Parameters: *loc* 3D coordinate.
radius To be searched.
mask Mask of object type to look for.
Return: *nothing*
Description: Searches for objects of type *mask* within a radius around the location.
Usage: `InitContainerRadiusSearch("0 450 76", %somerad, DebrisObjectType);`

InputLog(filename)

Parameters: *filename* String containing file name.
Return: *nothing*
Description: *Windows Only:* Enables or disables logging of `DirectInput` events to log file specified by string.
Usage: `InputLog(DI.log);`

IsDemoRecording()

Parameters: *none*
Return: *numeric* 1 (or `true`) = enable, 0 (or `false`) = disable.
Description: Queries if a demo is currently being recorded.
Usage: `%state = IsDemoRecording();`

IsDeviceFullScreenOnly(devicename)

Parameters:	*devicename*	Name of device to query
Return:	*numeric*	1 = yes, 0 = no.
Description:	Queries if device is capable of full screen only.	
Usage:	`IsDeviceFullScreenOnly(%devicename);`	

IsEventPending(%id)

Parameters:	*id*	ID number to check
Return:	*numeric*	1 = `true`, 0 = `false`.
Description:	Queries if an event is pending with an ID number of *id*.	
Usage:	`%status = IsEventPending($eventid);`	

IsFile(filename)

Parameters:	*filename*	String containing full file name.
Return:	*numeric*	1 = `true`, 0 = `false`.
Description:	Queries if the file exists in the Torque Script file name buffer.	
Usage:	`%result = IsFile("/common/server/script.cs");`	

IsFullScreen()

Parameters:	*none*	
Return:	*numeric*	1 = yes, 0 = no.
Description:	Queries whether screen mode is set to full screen.	
Usage:	`%result = IsFullScreen();`	

IsJoystickDetected()

Parameters:	*none*	
Return	*numeric*	1 = `true`, 0 = `false`.
Description:	Determines if a joystick is present.	
Usage:	`%jd = IsJoystickDetected();`	

IsKoreanBuild()

Parameters: *none*
Return: *string*
Description: Korean registry key checker.
Usage:
```
%kb = IsKoreanBuild();
```

IsObject(handle)

Parameters: *handle* Handle of supposed object.
Return: *numeric* 1 = `true`, 0 = `false`.
Description: Queries if *handle* is an object.
Usage:
```
%status = IsObject(%chopper);
```

IsPackage(name)

Parameters: *name* String containing the name of the package.
Return: *numeric* 1 = `true`, 0 = `false`.
Description: Queries if *name* is a registered package.
Usage:
```
%status = IsPackage(Show);
```

IsPointInside(point)

Parameters: *point* "x y".
Return: *numeric* 1 = `true`, 0 = `false`.
Description: Queries if *point* is coincident with the interior of any object.
Usage:
```
%status = IsPointInside("123 345 25");
```

IsWriteableFileName(filename)

Parameters: *filename* String containing full file name.
Return: *numeric* 1 = `true`, 0 = `false`.
Description: Queries if file specified by *filename* is writeable.
Usage:
```
%result = IsWriteableFileName("/common/server/script.cs");
```

Team LRN

LaunchDedicatedServer(missionType, map, botCount)

Parameters:	*missionType*	Game- or Mod-specific string.
	map	Mission or map name string.
	botCount	Number of AI bots allowed.
Return:	*nothing*	
Description:	Starts dedicated game server with specified arguments.	
Usage:	`LaunchDedicatedServer(mymission,damap,0);`	

LightScene(completion)

Parameters:	*completion*	Completion callback.
Return:	*numeric*	Function handle
Description:	Lights the current mission using the callback function pointed to by *completion* when mission lighting is finished.	
Usage:	`%result = LightScene("CompletionCallback")`	

lockMouse(switch)

Parameters:	*switch*	1 (or `true`) = lock, 0 (or `false`) = unlock.
Return:	*nothing*	
Description:	Toggles the mouse state.	
Usage:	`lockMouse(true);`	

ltrim(str)

Parameters:	*str*	String to be processed.
Return:	*string*	The processed resultant string.
Description:	Strips any white space from **str** from the left side (before any other characters) of *str*. White space is defined as space, carriage returns, or new line characters.	
Usage:	`%tidystring = ltrim(%yuckystring);`	

mAbs(x)

Parameters:	*x*	Operand. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the absolute value of x.	
Usage:	`%val = mAbs(76.3);`	

mAcos(x)

Parameters:	*x*	Radian. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the arc cosine.	
Usage:	`%val = mAcos(2.0);`	

makeTestTerrain(filename)

Parameters:	*filename*	String containing file name.
Return:	*nothing*	
Description:	Makes a test terrain file.	
Usage:	`makeTestTerrain("testfile");`	

mAsin(x)

Parameters:	*x*	Radian. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the arc sine.	
Usage:	`%val = mAsin(1.5);`	

mAtan(x,y)

Parameters:	*x*	Radian. Can be an integer or a floating point.
	y	Radian. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the arc tangent.	
Usage:	`%val = mAtan(-1.667,2);`	

MathInit(mode)

Parameters:	*mode*	The string specifier. Choices: "DETECT" "C" "FPU" "MMX" "3DNOW" "SSE"
Return:	*nothing*	
Description:	Enables math extensions based on CPU type.	
Usage:	`MathInit("DETECT");`	

MatrixCreate(vector, angledvector)

Parameters:	*vector*	"x y z".
	angledvector	"x y z angle".
Return:	*string*	
Description:	Generates a matrix from the specified values.	
Usage:	`%mtx = MatrixCreate("10 10 30", "30 40 50 10");`	

MatrixCreateFromEuler (valstring)

Parameters:	*valstring*	"x y z".
Return:	*string*	
Description:	Generates a matrix from given arguments.	
Usage:	`%val = MatrixCreateFromEuler("5.5 90 200");`	

MatrixMulPoint(matrix, point)

Parameters:	*matrix*	
	point	
Return:	*string*	
Description:	Multiplies a matrix by a point.	
Usage:	`%mtx = MatrixMulPoint(%matrix,%point);`	

MatrixMultiply(matrixA, matrixB)

Parameters: *matrixA*
matrixB
Return: *string*
Description: Multiplies two matrices.
Usage:
```
%mtx = MatrixMultiply(matrix1,matrix2);
```

MatrixMulVector(matrix, vector)

Parameters: *matrix*
vector
Return: *string*
Description: Multiplies a matrix by a vector.
Usage:
```
%mtx = MatrixMulVector(matrix,vector);
```

mCeil(x)

Parameters: *x* Operand. Can be an integer or a floating point.
Return: *numeric*
Description: Finds the smallest integral value greater than or equal to the operand.
Usage:
```
%val = mCeil(%dialogHeight / %textHeight);
```

mCos(x)

Parameters: *x* Radian. Can be an integer or a floating point.
Return: *numeric*
Description: Computes the cosine.
Usage:
```
%val = mCos(69);
```

mDegToRad(degrees)

Parameters: *degrees* Degrees to be converted. Can be an integer or a floating point.
Return: *numeric*
Description: Converts degrees to radians.
Usage:
```
%rads = mDegToRad(90);
```

mFloatLength(x, len)

Parameters:	*x*	Operand. Can be an integer or a floating point.
	len	Number of decimal places.
Return:	*numeric*	
Description:	Returns x as a floating point value with *len* decimal places.	
Usage:	`%mypi = mFloatLength((21/7),8);`	

mFloor(x)

Parameters:	*x*	Operand. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Finds the largest integral value less than or equal to the operand.	
Usage:	`%val = mFloor(%dialogHeight / %textHeight);`	

mLog(x)

Parameters:	*x*	Radian. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the natural logarithm.	
Usage:	`%val = mLog(7654.98);`	

mPow(x,y)

Parameters:	*x*	Base. Can be an integer or a floating point.
	y	Exponent. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes *x* raised to the power of *y*.	
Usage:	`%val = mPow(2,4);`	

mRadToDeg(radians)

Parameters:	*radians*	Radians to be converted. Can be integers or floating points.
Return:	*numeric*	
Description:	Converts radians to degrees.	
Usage:	`%degs = mRadToDeg(1);`	

msg(handle,message)

Parameters:	*handle*	Handle of object to receive message.
	message	String containing message.
Return:	*nothing*	
Description:	Sends *message* to the object specified by *handle*.	
Usage:	`msg(%objhandle, %msg);`	

mSin(x)

Parameters:	*x*	Radian. Can be an integer or a floating point.
Return:	*numeric*	
Description:	Computes the sine.	
Usage:	`%val = mSin(65);`	

mSolveCubic(a,b,c,d)

Parameters:	*a,b,c,d*	Operands. Can be integers or floating points.
Return:	*string*	
Description:	Computes a cubic solution for *x*. a*x*^3 + b*x*^2 + c*x* + d = 0.	
Usage:	`%val = mSolveCubic(a,b,c,d);`	

mSolveQuadratic(a,b,c)

Parameters:	*a,b,c*	Operands. Can be integers or floating points.
Return:	*string*	
Description:	Computes a quadratic solution for *x*. a*x*^2 + b*x* + c = 0.	
Usage:	`%val = mSolveQuadratic(a,b,c);`	

mSolveQuartic(a,b,c,d,e)

Parameters:	*a,b,c,d,e*	Operands. Can be integers or floating points.
Return:	*string*	
Description:	Computes a quartic solution for *x*. a*x*^4 + b*x*^3 + c*x*^2 + d*x* + e = 0.	
Usage:	`%val = mSolveQuartic(a,b,c,d,e);`	

mSqrt(x)

Parameters: *x* Operand. Can be an integer or a floating point.
Return: *numeric*
Description: Computes the square root of *x*.
Usage:
```
%val = mSqrt(81);
```

mTan(x)

Parameters: *x* Radian. Can be an integer or a floating point.
Return: *numeric*
Description: Computes the tangent.
Usage:
```
%val = mTan(45.0);
```

nameToID(name)

Parameters: *name* String containing the name of the object.
Return: *nothing*
Description: Gets the ID number of the named object.
Usage:
```
nameToID(%chopper);
```

nextResolution()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Increases next highest resolution.
Usage:
```
%result = nextResolution ();
```

nextToken (str,token,delim)

Parameters:	*str*	Initializes tokenizer when set to a valid string variable. Uses an empty string ("") to specify follow-up operation on the same string.
	token	Reference handle to the variable that will receive the found token. *Note:* When passing a variable by reference to a function, such as with this parameter, you do not prefix the variable name with % or $.
	delim	Specifies the character that delimits the tokens.
Return:	*string*	Balance of the string after the found token.
Description:	Sets *token* to the next substring in *str* delimited by *delim*. The initial call to this function specifies *str*; subsequent calls to this function that operate on the same string must pass the empty string ("").	
Usage:	`%str = nextToken("one,two,three", number, ",");`	

OpenALInitDriver()

Parameters:	*none*
Return:	*numeric*
Description:	Initializes the sound driver.
Usage:	`OpenALInitDriver();`

OpenALShutdownDriver()

Parameters:	*none*
Return:	*nothing*
Description:	Disables the sound driver.
Usage:	`OpenALShutdownDriver();`

PanoramaScreenShot(filename)

Parameters:	*filename*	String containing file name.
Return:	*nothing*	
Description:	Captures the panoramic screen view and saves it to the file specified by *filename*. The engine will take the panoroma shot as a sequence of three screen captures, looking left, center, then right.	
Usage:	`PanoramaScreenShot("myPanorama");`	

pathOnMissionLoadDone()

Parameters: *none*
Return: *nothing*
Description: Sets the Mod path that will be active when a mission is finished loading.
Usage:
```
pathOnMissionLoadDone("missE/mission");
```

PermDisableMouse()

Parameters: *none*
Return: *nothing*
Description: Permanently disables `DirectInput` polling of the mouse device.
Usage:
```
PermDisableMouse();
```

PlayDemo(filename)

Parameters: *filename* String containing file name.
Return: *nothing*
Description: Plays back a demo saved in *filename*.
Usage:
```
PlayDemo(MyNiftyDemo);
```

PlayJournal(name,[break])

Parameters: *name* String containing file name of journal.
break If `true`, then stops playback after each event.
Return: *nothing*
Description: Plays back saved journal specified by *name*.
Usage:
```
PlayJournal("myjrnl.jnl");
```

PrevResolution()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Decreases next highest resolution.
Usage:
```
%result = PrevResolution();
```

ProfilerDump()

Parameters: *none*
Return: *nothing*
Description: Dumps `NetStringTable` statistics to the console.
Usage:
```
ProfilerDump();
```

ProfilerDumpToFile(filename)

Parameters: *filename* String containing file name.
Return: *nothing*
Description: Dumps `NetStringTable` statistics to the file specified by *`filename`*.
Usage:
```
ProfilerDumpToFile(dump.txt);
```

ProfilerEnable(switch)

Parameters: *switch* 1 enables, 0 disables.
Return: *nothing*
Description: Enables or disables profiling.
Usage:
```
ProfilerEnable(false);
```

ProfilerMarkerEnable(markerName, switch)

Parameters: *markerName* Name of profile marker.
switch 1 enables, 0 disables.
Return: *nothing*
Description: Enables or disables profiling for `markerName`.
Usage:
```
ProfilerMarkerEnable(mark,true);
```

PurgeResources()

Parameters: *none*
Return: *nothing*
Description: Purges all resources used by the game through the resource manager.
Usage:
```
PurgeResources();
```

Team LRN

QueryMasterServer (port,flags,gametype,missiontype,minplayers,maxplayers, maxbots,regionmask,maxping,filterflags,mincpu,*buddycount*,*buddylist*)

Parameters:	*port*	Master server port.
	flags	The query flags. Choices:
		0x00 = online query
		0x01 = offline query
		0x02 = no string compression
	gametype	Game type string.
	missiontype	Mission type string.
	minplayers	Minimum number of players for viable game.
	maxplayers	Maximum allowable players.
	maxbots	Maximum allowable connected AI bots.
	regionmask	Numeric discriminating mask.
	maxping	Maximum ping for connecting clients; 0 means no maximum.
	filterflags	Server filters. Choices:
		0x00 = dedicated
		0x01 = not password protected
		0x02 = Linux
		0x80 = current version
	mincpu	Minimum specified CPU capability.
	buddycount	Number of buddy servers in buddy list.
	buddylist	List of server names that are buddies to this server.
Return:	*nothing*	
Description:	Queries a master server looking for specified information. The responses are accessible from the `ServerList` array.	
Usage:	`QueryMasterServer(`	
	`28000, 0, $Client::GameTypeQuery, Client::MissionTypeQuery,`	
	`0, 100, 0, 2, 0, 100, 0, 0, "" );`	

Quit()

Parameters:	*none*
Return:	*nothing*
Description:	Quits the game.
Usage:	`Quit();`

RedbookClose()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Closes the currently open redbook (CD) device.
Usage:
```
%result = RedbookClose();
```

RedbookGetDeviceCount()

Parameters: *none*
Return: *numeric*
Description: Queries for the number of redbook (CD) devices.
Usage:
```
%count = RedbookGetDeviceCount();
```

RedbookGetDeviceName(idx)

Parameters: *idx* Device index.
Return: *string*
Description: Queries the device name of redbook (CD) at the specified device index.
Usage:
```
%name = RedbookGetDeviceName(1);
```

RedbookGetLastError()

Parameters: *none*
Return: *string*
Description: Queries for the last error from a redbook (CD) device.
Usage:
```
%error = RedbookGetLastError();
```

RedbookGetTrackCount()

Parameters: *none*
Return: *numeric*
Description: Queries the number of redbook (CD) tracks.
Usage:
```
%tracks = RedbookGetTrackCount();
```

Team LRN

RedbookGetVolume()

Parameters: *none*
Return: *numeric*
Description: Queries the current volume level of a redbook (CD) device.
Usage:
```
%volume = RedbookGetVolume();
```

RedbookOpen([name])

Parameters: *name* If non-null, specifies the device.
Return: *numeric* 1 = success, 0 = fail.
Description: Opens a redbook (CD) device.
Usage:
```
%result = RedbookOpen();
```

RedbookPlay(track)

Parameters: *track* Index of track.
Return: *numeric* 1 = success, 0 = fail.
Description: Plays a track on a redbook (CD) device.
Usage:
```
%result = RedbookPlay(2);
```

RedbookSetVolume(volume)

Parameters: *volume* Volume setting.
Return: *numeric* 1 = success, 0 = fail.
Description: Sets the volume of a redbook (CD) device.
Usage:
```
%result = RedbookSetVolume(%volume);
```

RedbookStop()

Parameters: *none*
Return: *numeric* 1 = success, 0 = fail.
Description: Stops playing on the current redbook (CD) device.
Usage:
```
%result = RedbookStop();
```

Team LRN

RemoveField(text, index)

Parameters:	*text*	String with field-delimited words.
	index	Field-based offset into the text string.
Return:	*string*	The resultant string.
Description:	Removes the field-string at *index* from *text.*	
Usage:	`%result = RemoveField(%text, 0);`	

RemoveRecord (text, index)

Parameters:	*text*	String with new line-delimited records.
	index	Record-based offset into the text string.
Return:	*string*	The resultant string.
Description:	Removes the record-string at *index* from *text*.	
Usage:	`%str = RemoveRecord(%text, 0);`	

RemoveTaggedString(tag)

Parameters:	*tag*	Numeric tag of string to be removed.
Return:	*nothing*	
Description:	Removes a tagged string from the list.	
Usage:	`RemoveTaggedString(%tagname);`	

RemoveWord(text, index)

Parameters:	*text*	String with space-delimited words.
	index	Word-based offset into the text string.
Return:	*string*	The resultant string.
Description:	Removes the word-string at *index* from *text*.	
Usage:	`%str = RemoveWord(%text, 0);`	

ResetLighting()

Parameters:	*none*
Return:	*nothing*
Description:	Resets the current lighting.
Usage:	`ResetLighting();`

Team LRN

RestWords(text)

Parameters:	*text*	String with space-delimited words.
Return:	*string*	The resultant string.
Description:	Returns the words remaining after the first word in *text*.	
Usage:	`%data = RestWords(%text);`	

Rtrim(str)

Parameters:	*str*	String to be processed.
Return:	*string*	The processed resultant string.
Description:	Strips any white space from *str* from the right side (after all other characters) of *str*. White space is defined as space, carriage returns, or new line characters.	
Usage:	`%tidystring = Rtrim(%yuckystring);`	

SaveJournal(name)

Parameters:	*name*	String containing file name of journal.
Return:	*nothing*	
Description:	Save a journal to file specified by *name*.	
Usage:	`SaveJournal("myjrnl.jnl");`	

Schedule(time, reference, command, <arg1...argN>)

Parameters:	*time*	Time to wait for trigger, in milliseconds.
	reference	Handle of object to attach schedule to, or 0.
	command	Command to execute.
	arg1…argN	Arguments to accompany command. (optional)
Return:	*numeric*	Event ID.
Description:	Schedules an event that will trigger in *time* milliseconds and execute *command*, with *args*. If *reference* is not 0, then it must be a valid object handle. If the object is deleted, the scheduled event is discarded.	
Usage:	`$evt = Schedule(5000, 0, "updateRadar");`	

ScreenShot(filename)

Parameters:	*filename*	String containing file name.
Return:	*nothing*	
Description:	Captures the screen view and saves it to file specified by *filename*.	
Usage:	`ScreenShot("myScreen");`	

Team LRN

SetDefaultFov(fov)

Parameters:	*fov*	Numeric in degrees.
Return:	*nothing*	
Description:	Sets the default field of view.	
Usage:	`SetDefaultFov( 60 );`	

SetDisplayDevice(deviceName[, width[, height[, bpp[, fullScreen]]]])

Parameters:	*deviceName*	Name of target device driver.
	width	Screen width.
	height	Screen height.
	bpp	Bits per pixel.
	fullScreen	1 enables, 0 disables.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Sets up the display device with specified values.	
Usage:	`%result = SetDisplayDevice ( "OpenGL", 800, 600, 32, true );`	

SetEchoFileLoads(switch)

Parameters:	*switch*	1 (or `true`) enables, 0 (or `false`) disables.
Return:	*nothing*	
Description:	Enables or disables File Load echo to console.	
Usage:	`SetEchoFileLoads(1);`	

SetField(text, index, subst)

Parameters:	*text*	String with field-delimited words.
	index	Field-based offset into the text string.
	subst	Substitute string.
Return:	*string*	The resultant string.
Description:	Substitutes the field-string *sub* for the word-string found at *index* in the string *text*.	
Usage:	`%rec = SetField(%text, 0, "blah");`	

SetFov(val)

Parameters:	*val*	The field of view (degrees).
Return:	*nothing*	
Description:	Sets the current field of view.	
Usage:	`SetFov(90);`	

SetFSAA(switch, level)

Parameters:	*switch*	1 enables, 0 disables.
	level	Target level.
Return:	*nothing*	
Description:	Invokes `SetNPatch` with the same parameters.	
`Usage:`	`SetFSAA(%newstate,%lvl)`	

SetInteriorFocusedDebug(which)

Parameters:	*which* Handle of interior for focus. If *which* has a value, then debugging is enabled; if *which* is empty (not passed), then debugging is disabled.
Return:	*nothing*
Description:	Enables debug mode for interior focused objects.
Usage:	`SetInteriorFocusedDebug();`

SetInteriorRenderMode(mode)

Parameters:	*mode*
Return:	*nothing*
Description:	Sets the detail render level for interiors.
Usage:	`SetInteriorRenderMode(7);`

SetLogMode(mode)

Parameters:	*mode*	The numeric mode value. Choices: 0 = no logging 1 = logging on, append mode 2 = logging on, overwrite mode
Return:	*nothing*	
Description:	*Enables or disables error logging to disk.*	
Usage:	`SetLogMode(1);`	

Team LRN

SetModPaths(path)

Parameters:	*path*	String containing path.
Return:	*nothing*	
Description:	Set Mod path. This specifies which folders will be visible to the scripts and the resource engine.	
Usage:	`SetModPaths("common;game");`	

SetNetPort(port)

Parameters:	*port*	Port number.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Sets the network port.	
Usage:	`%result = SetNetPort(1313);`	

SetNPatch(switch, level)

Parameters:	*switch*	1 enables, 0 disables.
	level	Target level.
Return:	*nothing*	
Description:	Enables or disables `Npatching` (quadratic interpolation) at specified *level.*	
Usage:	`SetNPatch(1, 1);`	

SetOpenGLAnisotropy(level)

Parameters:	*level*	0=trilinear, 1=bilinear.
Return:	*nothing*	
Description:	Sets the level of anisotropy.	
Usage:	`SetOpenGLAnisotropy(0);`	

SetOpenGLInteriorMipReduction(level)

Parameters:	*level*	Mipmap level (0 = minimum detail, 5 = maximum detail).
Return:	*nothing*	
Description:	Sets interior texture detail.	
Usage:	`SetOpenGLInteriorMipReduction(2);`	

Team LRN

SetOpenGLMipReduction(level)

Parameters:	*level*	Mipmap level (0 = minimum detail, 5 = maximum detail).
Return:	*nothing*	
Description:	Sets shape texture detail to *level.*	
Usage:	`SetOpenGLMipReduction(2);`	

SetOpenGLSkyMipReduction(level)

Parameters:	*level*	Mipmap level (0 = minimum detail, 5 = maximum detail).
Return:	*nothing*	
Description:	Sets skybox and cloud texture detail.	
Usage:	`SetOpenGLMipReduction(2);`	

SetOpenGLTextureCompressionHint(hint)

Parameters:	*hint*	The compression level hint. Choices: GL_DONT_CARE GL_FASTEST GL_NICEST
Return:	*nothing*	
Description:	Suggests texture compression mode.	
Usage:	`SetOpenGLTextureCompressionHint(GL_NICEST);`	

SetRandomSeed([seed])

Parameters:	*seed*	Starting point.
Return:	*nothing*	
Description:	Sets the current starting point for generating a series of pseudo-random numbers.	
Usage:	`SetRandomSeed();`	

SetRecord (text, index, subst)

Parameters:	*text*	String with new line-delimited records.
	index	Record-based offset into the text string.
	subst	Substitute string.
Return:	*string*	The resultant string.
Description:	Substitutes the record-string *sub* for the record-string found at *index* in the string *text.*	
Usage:	`%str = SetRecord(%text, 0, "blah");`	

Team LRN

SetResolution(width, height, bpp)

Parameters:	*width*	Screen width.
	height	Screen height.
	bpp	Bits per pixel.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Sets the screen resolution to specified values.	
Usage:	`%result = SetResolution(640,480,32);`	

SetScreenMode(width, height, bpp, fullScreen)

Parameters:	*width*	Screen width.
	height	Screen height.
	bpp	Bits per pixel.
	fullScreen	1 enables, 0 disables.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Sets up the screen with specified values.	
Usage:	`%result = SetScreenMode( 800, 600, 32, true );`	

SetServerInfo(index)

Parameters:	*index*	Row of interest in the server list.
Return:	*numeric*	1 = success, 0 = fail.
Description:	Changes our indexed reference into the `ServerList`.	
Usage:	`%result = SetServerInfo(%index);`	

SetShadowDetailLevel(level)

Parameters:	*level*	Numeric range 0.0 to 1.0.
Return:	*nothing*	
Description:	Sets the level of detail for shadows.	
Usage:	`SetShadowDetailLevel( 1.0 );`	

SetVerticalSync(switch)

Parameters:	*switch*	1 enables, 0 disables.
Return:	*numeric*	1 = `true`, 0 = `false`.
Description:	Enables or disables the use of `VerticalSync`.	
Usage:	`SetVerticalSync(true);`	

SetWord(text, index, subst)

Parameters:	*text*	String with space-delimited words.
	index	Word-based offset into the text string.
	subst	Substitute string.
Return:	*string*	The resultant string.
Description:	Substitutes the word-string *sub* for the word-string found at *index* in the string *text*.	
Usage:	`%str = SetWord(%text, 0, "blah");`	

SetZoomSpeed(speed)

Parameters:	*speed*	Transition speed. Ranges from 0 to 2,000 milliseconds.
Return:	*nothing*	
Description:	Sets the transition speed when changing field of view.	
Usage:	`SetZoomSpeed( speed );`	

StartHeartbeat()

Parameters:	*none*
Return:	*nothing*
Description:	Begins periodic messages to the master server that show that this server is still alive.
Usage:	`schedule(0,0,StartHeartbeat);`

StartRecording(filename)

Parameters:	*filename*	String containing file name.
Return:	*nothing*	
Description:	Records a demo and saves it as f*ilename*.	
Usage:	`StartRecording(myDemo);`	

StopHeartbeat()

Parameters:	*none*
Return:	*nothing*
Description:	Stops the heartbeat messages.
Usage:	`StopHeartbeat();`

StopRecording()

Parameters: *none*
Return: *nothing*
Description: Stops the currently recording demo.
Usage:
```
StopRecording();
```

StopServerQuery()

Parameters: *none*
Return: *nothing*
Description: Cancels the current query and marks outstanding pings as finished.
Usage:
```
StopServerQuery();
```

Strchr(str, char)

Parameters:	*str*	String to be processed.
	char	String containing the character to be found.
Return:	*string*	
Description:	Finds the first substring in the string that begins with *char*.	
Usage:	`%file = Strchr("data/file.dat", "/" );`	

Strcmp(str1, str2)

Parameters:	*str1*	First string.	
	str2	Second string.	
Return:	*numeric*	< 0	*str1* is less than (also not equal to) *str2*.
		0	*str1* is equal to *str2*.
		> 0	*str1* is greater than (also not equal to) *str2*.
Description:	Case-*sensitive* comparison of two strings: *str1* and *str2*.		

Usage:
```
if ( Strcmp( %weaponName, "candlestick" ) == 0 )
     return %weaponFound;
```

Stricmp(str1, str2)

Parameters:	*str1*	First string.	
	str2	Second string.	
Return:	*numeric*	< 0	*str1* is less than (not equal to) *str2*.
		0	*str1* is equal to *str2*.
		> 0	*str1* is greater than (not equal to) *str2*.
Description:	Case-*insensitive* comparison of two strings: *str1* and *str2*.		
Usage:	`if ( Stricmp(%weaponName, "CandleStick") == 0 )` `  return %weaponFound;`		

StripChars(str, chars)

Parameters:	*str*	String to be processed.
	chars	String containing characters to be stripped.
Return:	*string*	The processed resultant string.
Description:	Removes all characters in the string *chars* from the string *str*.	
Usage:	`%stripped = StripChars(%value, "~" );`	

StripMLControlChars(string)

Parameters:	*string*
Return:	*string*
Description:	Strips ML special control characters from string.
Usage:	`%text = StripMLControlChars(%string);`

StripTrailingSpaces(string)

Parameters:	*string*	Input string.
Return:	*string*	
Description:	Strips trailing spaces and underscores from string to be used for player name.	
Usage:	`%name = StripTrailingSpaces( strToPlayerName( %name ) );`	

Strlen(str)

Parameters:	*str*	String.
Return:	*numeric*	
Description:	Obtains the number of characters in *str.*	
Usage:	`%len = Strlen(%weaponName);`	

Team LRN

Strlwr(str)

Parameters:	*str*	String to be processed.
Return:	*string*	The processed resultant string.
Description:	Converts all characters in *str* to lowercase.	
Usage:	`%var = Strlwr(%value);`	

Strpos(str, target[, offset])

Parameters:	*str*	String to be searched.
	target	String to find.
	offset	Starts search at offset. (optional)
Return:	*numeric*	
Description:	Finds the first occurrence of the target string in the search string, with optional starting offset. *Note:* This function is identical to `strstr` when *offset* isn't used.	
Usage:	`%pos = Strpos(%weaponName, "gun") ;`	

Strreplace(str, target, subst)

Parameters:	*str*	String to be processed.
	target	Target string to be replaced.
	subst	Substitute string.
Return:	*string*	The processed resultant string.
Description:	Replaces all instances of *target* and replaces with *subst*.	
Usage:	`%dospath = Strreplace(%path, "/", "\");`	

Strstr(str, target)

Parameters:	*str*	String to be tested.
	target	Target substring to find.
Return:	*numeric*	Offset within *str* where *target* was found.
Description:	Finds first occurrence of a *target* within *string*.	
Usage:	`%loc = Strstr( %weaponName, "stick" );`	

StrToPlayerName(string);

Parameters:	*string*	Player name string.
Return:	*string*	
Description:	Converts name string to properly formatted player name string. Proper formatting means the player name is limited to 16 characters in length. Leading and trailing spaces are trimmed; reserved characters are removed.	
Usage:	`%newname = StrToPlayerName( %name );`	

Strupr(str)

Parameters:	*str*	String to be processed.
Return:	*string*	The processed resultant string.
Description:	Converts all characters in *str* to uppercase.	
Usage:	`%var = Strupr(%value);`	

SwitchBitDepth()

Parameters:	*none*	
Return:	*numeric*	1 = success, 0 = fail.
Description:	Switches between 16 and 32 bits per pixel in full-screen mode.	
Usage:	`%result = SwitchBitDepth();`	

TelnetSetParameters(port, consolePW, listenPW)

Parameters:	*port*	Connection port.
	consolePW	Console password.
	listenPW	"Listener" password.
Return:	*nothing*	
Description:	Initializes telnet connection request parameters.	
Usage:	`TelnetSetParameters(4123, "garage", "games");`	

ToggleFullScreen()

Parameters:	*none*	
Return:	*numeric*	1 = success, 0 = fail.
Description:	Switches between windowed mode and full-screen mode.	
Usage:	`%result = ToggleFullScreen();`	

Team LRN

ToggleInputState()

Parameters: *none*
Return: *nothing*
Description: Toggles `DirectInput` state between *enabled* and *disabled*. Also prints the new input state (same as `echoInputState`) to the console.
Usage:
```
ToggleInputState();
```

ToggleNPatch()

Parameters: *none*
Return: *nothing*
Description: Toggles the enable/disable state of `Npatch`.
Usage:
```
ToggleNPatch();
```

Trace(switch)

Parameters: *switch* 1 (or `true`) enables, 0 (or `false`) disables.
Return: *nothing*
Description: Turns execution trace on or off.
Usage:
```
Trace(true);
```

Trim(str)

Parameters: *str* String to be processed.
Return: *string* The processed resultant string.
Description: Strips any white space from *str* from the left or right sides (before and after all other characters) of *str*. White space is defined as space, carriage returns, or new line characters.
Usage:
```
%tidystring = Trim(%yuckystring);
```

ValidateMemory()

Parameters: *none*
Return: *nothing*
Description: Ensures sufficient memory available for the program.
Usage:
```
ValidateMemory();
```

Team LRN

VectorAdd(vector1, vector2)

Parameters: *vector1* "x y z".
vector2 "x y z".
Return: *string*
Description: Adds *vector2* to *vector1*.
Usage:
```
%result = VectorAdd("87.21 54.11 10.0", "9.99 12.6 6.00");
```

VectorCross(vector1, vector2)

Parameters: *vector1* "x y z".
vector2 "x y z".
Return: *string*
Description: Computes the cross product between two vectors.
Usage:
```
%product = VectorCross("x y z","x y z");
```

VectorDist(vector1, vector2)

Parameters: *vector1* "x y z".
vector2 "x y z".
Return: *string*
Description: Computes the distance between two vectors.
Usage:
```
%delta = VectorDist(%vector1, %vector2);
```

VectorDot(vector1, vector2)

Parameters: *vector1* "x y z".
vector2 "x y z".
Return: *string*
Description: Computes the dot product between two vectors.
Usage:
```
%product = VectorDot("0 0 1",%eye);
```

VectorLen(vector)

Parameters: *vector* "x y z".
Return: *string*
Description: Computes the length of the vector.
Usage:
```
%len = VectorLen(vector);
```

Team LRN

VectorNormalize(vector)

Parameters:	*vector*	"x y z".
Return:	*string*	
Description:	Normalizes a vector.	
Usage:	`%nvector = VectorNormalize("5 10 30");`	

VectorOrthoBasis(vector)

Parameters:	*vector*	"x y z".
Return:	*string*	
Description:	Computes the orthogonal normal for a vector.	
Usage:	`%normal = VectorOrthoBasis("x y z angle");`	

VectorScale(vector, scalar)

Parameters:	*vector*	"x y z".
	scalar	Can be an integer or a floating point.
Return:	*string*	
Description:	Computes the result of the vector sized by the scale.	
Usage:	`%svector = VectorScale("5 10 30", 100);`	

VectorSub(vector1, vector2)

Parameters:	*vector1*	"x y z".
	vector2	"x y z".
Return:	*string*	
Description:	Subtracts *vector2* from *vector1*.	
Usage:	`%result = VectorSub("34.0989 989.3249 100.00", %position);`	

VideoSetGammaCorrection(gamma)

Parameters:	*gamma*	Gamma correction setting.
Return:	*nothing*	
Description:	Sets the gamma correction.	
Usage:	`VideoSetGammaCorrection(0.5);`	

Team LRN

Warn(text)

Parameters:	*text*	String.
Return:	*nothing*	
Description:	Prints *text* to the console with light gray font. Text can be formatted according to the string rules.	
Usage:	`Warn("Danger, Will Robinson!!");`	

Torque Reference Tables

Table A.1 Torque Script Object Type Masks

Mask Identifier	Number	Mask Bit Position
DefaultObjectType	0	0
StaticObjectType	1	1
EnvironmentObjectType	2	2
TerrainObjectType	4	3
InteriorObjectType	8	4
WaterObjectType	16	5
TriggerObjectType	32	6
MarkerObjectType	64	7
unassigned	128	8
unassigned	256	9
DecalManagerObjectType	512	10
GameBaseObjectType	1024	11
ShapeBaseObjectType	2048	12
CameraObjectType	4096	13
StaticShapeObjectType	8192	14
PlayerObjectType	16384	15
ItemObjectType	32768	16
VehicleObjectType	65536	17
VehicleBlockerObjectType	131072	18
ProjectileObjectType	262144	19
ExplosionObjectType	524288	20
unassigned	2097152	21
CorpseObjectType	1048576	22
DebrisObjectType	4194304	23
PhysicalZoneObjectType	8388608	24
unassigned	33554432	25
StaticTSObjectType	16777216	26

continued

Team LRN

StaticRenderedObjectType	67108864	27
unassigned	268435456	28
unassigned	536870912	29
unassigned	1073741824	30
unassigned	2147483648	31

Table A.2 Torque Object Methods

Object Class	Method
AIPlayer	ai.moveForward()
	ai.walk()
	ai.run()
	ai.stop()
	ai.setMoveSpeed(float)
	ai.setTargetObject(object)
	ai.getTargetObject()
	ai.targetInSight()
	ai.aimAt(point)
	ai.getAimLocation()
BanList	BanList::add(id, TA, banTime)
	BanList::addAbsolute(id, TA, banTime)
	BanList::removeBan(id, TA)
	BanList::isBanned(id, TA)
	BanList::export(filename)
Camera	camera.getPosition()
	camera.setOrbitMode(obj, xform,min-dist,max-dist, cur-dist)
	camera.setFlyMode()
	watchView.edit(newValue)
	watchView.remove()
	watchView.queryAll()
	watchView.clear()
Debris	obj.init(position, velocity)
EditTSCtrl	EditTSCtrl.renderSphere(pos, radius,=)
	EditTSCtrl.renderCircle(pos, normal, radius,=)
	EditTSCtrl.renderTriangle(pnt, pnt, pnt)
	EditTSCtrl.renderLine(start, end)
FileObject	file.openForRead(fileName)
	file.openForWrite(fileName)
	file.openForAppend(fileName)
	file.writeLine(text)

continued

	file.isEOF()
	file.readLine()
	file.close()
FlyingVehicle	FlyingVehicle.setCreateHeight(bool)
GameBase	obj.getDataBlock()
	obj.setDataBlock(DataBlock)
GameConnection	conn.chaseCam(size)
	conn.setControlCameraFov(fov)
	conn.getControlCameraFov()
	conn.transmitDataBlocks(seq)
	conn.activateGhosting()
	conn.resetGhosting()
	conn.setControlObject(%obj)
	conn.getControlObject()
	conn.isAIControlled()
	conn.play2D(AudioProfile)
	conn.play3D(AudioProfile,Transform)
	conn.isScopingCommanderMap()
	conn.scopeCommanderMap(bool)
	conn.listenEnabled()
	conn.getListenState(clientId)
	conn.canListen(clientId)
	conn.listenTo(clientId, true|false)
	conn.listenToAll()
	conn.listenToNone()
	conn.setVoiceChannels(0-3)
	conn.setVoiceDecodingMask(mask)
	conn.setVoiceEncodingLevel(codecLevel)
	conn.setBlackOut(fadeTOBlackBool, timeMS)
	conn.setMissionCRC(crc)
GuiBitmapCtrl	guiBitmapCtrl.setBitmap(blah)
	guiBitmapCtrl.setValue(xAxis, yAxis)
GuiCanvas	canvas.renderFront(bool)
	canvas.setContent(ctrl)
	canvas.getContent()
	canvas.pushDialog(ctrl)
	canvas.popDialog()
	canvas.popLayer()
	canvas.cursorOn()
	canvas.cursorOff()
	canvas.setCursor(cursor)

continued

	canvas.hideCursor()
	canvas.showCursor()
	canvas.repaint()
	canvas.reset()
	canvas.isCursorOn()
	canvas.getCursorPos()
	canvas.setCursorPos(pos)
GuiControl	ctrl.getPosition()
	ctrl.getExtent()
	ctrl.getMinExtent()
	ctrl.resize(x, y, w, h)
	ctrl.setValue(value)
	ctrl.getValue()
	ctrl.setActive(value)
	ctrl.isActive()
	ctrl.setVisible(value)
	ctrl.isVisible()
	ctrl.isAwake()
	ctrl.setProfile(profile)
	ctrl.makeFirstResponder(value)
GuiEditCtrl	editCtrl.addNewCtrl(ctrl)
	editCtrl.select(ctrl)
	editCtrl.setRoot(root)
	editCtrl.setCurrentAddSet(ctrl)
	editCtrl.toggle()
	editCtrl.justify(mode)
	editCtrl.bringToFront()
	editCtrl.pushToBack()
	editCtrl.deleteSelection()
GuiFilterCtrl	guiFilterCtrl.getValue()
	guiFilterCtrl.setValue(f1, f2,...)
	guiFilterCtrl.identity()
GuiFrameSetCtrl	gfsc.frameBorder(index, enable)
	gfsc.frameMovable(index, enable)
	gfsc.frameMinExtent(index, w, h)
GuiInspector	inspector.inspect(obj)
	inspector.apply(newName)
GuiMessageVectorCtrl	[GuiMessageVectorCtrl].attach(MessageVectorId)
	[GuiMessageVectorCtrl].detach()
GuiPopUpMenuCtrl	menu.sort()
	menu.add(name,idNum{,scheme})

continued

Team LRN

	menu.addScheme(id, fontColor, fontColorHL, fontColorSEL)
	menu.getText()
	menu.setText(text)
	menu.getValue()
	menu.setValue(text)
	menu.clear()
	menu.forceOnAction()
	menu.forceClose()
	menu.getSelected()
	menu.setSelected(id)
	menu.getTextById(id)
	menu.setEnumContent(class, enum)
	menu.findText(text)
	menu.size()
	menu.replaceText(bool)
GuiSliderCtrl	guiSliderCtrl.getValue()
GuiTerrPreviewCtrl	guiTerrPreviewCtrl.reset()
	guiTerrPreviewCtrl.setRoot()
	guiTerrPreviewCtrl.getRoot()
	guiTerrPreviewCtrl.setOrigin(x, y)
	guiTerrPreviewCtrl.getOrigin()
	guiTerrPreviewCtrl.getValue()
	guiTerrPreviewCtrl.getValue(t)
GuiTextListCtrl	textList.getSelectedId()
	textList.setSelectedById(id)
	textList.setSelectedRow(index)
	textList.clearSelection()
	textList.clear()
	textList.addRow(id,text,index)
	textList.setRow(id,text)
	textList.getRowId(index)
	textList.removeRowById(id)
	textList.getRowTextById(id)
	textList.getRowNumById(id)
	textList.getRowText(index)
	textList.removeRow(index)
	textList.rowCount()
	textList.scrollVisible(index)
	textList.sort(colId{,increasing})
	textList.sortNumerical(colId{, increasing})
	textList.findText(text)

continued

	textList.setRowActive(id)
	textList.isRowActive(id)
GuiTreeViewCtrl	treeViewCtrl.open(obj)
HTTPObject	obj.get(addr, request-uri)
	obj.post(addr, request-uri, query, post)
InteriorInstance	[InteriorObject].activateLight()
	[InteriorObject].deactivateLight()
	[InteriorObject].echoTriggerableLights()
	[InteriorObject].getNumDetailLevels()
	[InteriorObject].setDetailLevel(level)
Item	obj.isStatic()
	obj.isRotating()
	obj.setCollisionTimeout(object)
	obj.getLastStickyPos()
	obj.getLastStickyNormal()
Lightning	[LightningObject].warningFlashes()
	[LightningObject].strikeRandomPoint()
	[LightningObject].strikeObject(id)
MessageVector	[MessageVector].deleteLine(DeletePos)
	[MessageVector].clear()
	[MessageVector].dump(filename{, header})
	[MessageVector].getNumLines()
	[MessageVector].getLineText(Line)
	[MessageVector].getLineTag(Line)
	[MessageVector].getLineTextByTag(Tag)
	[MessageVector].getLineIndexByTag(Tag)
PhysicalZone	obj.activate()
	obj.deactivate()
Player	obj.setActionThread(sequenceName)
	obj.setControlObject(obj)
	obj.getControlObject()
	obj.clearControlObject()
	obj.getDamageLocation(pos)
Precipitation	precipitation.setPercentage(percentage <1.0 to 0.0>)
	precipitation.stormPrecipitation(percentage <0 to 1>, Time)
SceneObject	obj.getScale()
	obj.getWorldBox()
	obj.getWorldBoxCenter()
	obj.getObjectBox()
	obj.getForwardVector()
ShapeBase	obj.setShapeName(tag)

continued

Team LRN

```
obj.getShapeName()
obj.playAudio(slot,AudioProfile)
obj.playAudio(slot)
obj.playThread(thread)
obj.setThreadDir(thread,bool)
obj.stopThread(thread)
obj.pauseThread(thread)
obj.mountObject(object,node)
obj.unmountObject(object)
obj.unmount()
obj.isMounted()
obj.getObjectMount()
obj.getMountedObjectCount()
obj.getMountedObjectNode(index)
obj.getMountedObject(index)
obj.getMountNodeObject(node)
obj.mountImage(DataBlock,slot,[loaded=true],[skinTag])
obj.unmountImage(slot)
obj.getMountedImage(slot)
obj.getPendingImage(slot)
obj.isImageFiring(slot)
obj.isImageMounted(DataBlock)
obj.getMountSlot(DataBlock)
obj.getImageSkinTag(slot)
obj.getImageState(slot)
obj.getImageTrigger(slot)
obj.setImageTrigger(slot,bool)
obj.getImageAmmo(slot)
obj.setImageAmmo(slot,bool)
obj.getImageTarget(slot)
obj.setImageTarget(slot,bool)
obj.getImageLoaded(slot)
obj.setImageLoaded(slot,bool)
obj.getMuzzleVector(slot)
obj.getMuzzlePoint(slot)
obj.getSlotTransform(slot)
obj.getAIRepairPoint()
obj.getVelocity()
obj.setVelocity(Vector)
obj.applyImpulse(Pos,Vector)
obj.getEyeVector()
```

continued

	obj.getEyeTransform()
	obj.setEnergyLevel(value)
	obj.getEnergyLevel()
	obj.getEnergyPercent()
	obj.setDamageLevel(value)
	obj.getDamageLevel()
	obj.getDamagePercent()
	obj.setDamageState(state)
	obj.getDamageState()
	obj.isDestroyed()
	obj.isDisabled()
	obj.isEnabled()
	obj.applyDamage(value)
	obj.applyRepair(value)
	obj.setRepairRate(value)
	obj.getRepairRate()
	obj.setRechargeRate(value)
	obj.getRechargeRate()
	obj.getControllingClient()
	obj.getControllingObject()
	obj.canCloak()
	obj.setCloaked(true|false)
	obj.isCloaked()
	obj.setDamageFlash(flash level)
	obj.getDamageFlash()
	obj.setWhiteOut(flash level)
	obj.getWhiteOut()
	obj.setInvincibleMode(time, speed)
	obj.getCameraFov()
	obj.setCameraFov(fov)
	obj.hide(bool)
	obj.isHidden()
	startFade(U32, U32,bool)
	obj.setDamageVector(vec)
ShapeBaseData	obj.checkDeployPos(xform)
	obj.getDeployTransform(pos, normal)
SimpleNetObject	obj.setMessage(msg)
Sky	sky.stormCloudsOn(0 or 1,Time)
	sky.stormFogOn(Percentage <0 to 1>, Time)
	sky.realFog(0 or 1, max, min, speed)
	sky.getWindVelocity()

continued

Team LRN

	sky.setWindVelocity(x, y, z)
	sky.stormCloudsShow(bool)
	sky.stormFogShow(bool)
StaticShape	obj.setPoweredState(bool)
	obj.getPoweredState(bool)
TCPObject	obj.listen(port)
	obj.send(string,...)
	obj.connect(addr)
	obj.disconnect()
Terraformer	Terraformer.canyon(dst, freq, turb, seed)
	Terraformer.preview(dst_gui, src)
	Terraformer.previewScaled(dst_gui, src)
	Terraformer.clearRegister(r)
	Terraformer.fBm(r, freq, 0.0-1.0{roughness}, detail, seed)
	Terraformer.smoothRidges(src,dst,0-1{factor},iterations)
	Terraformer.setTerrain(r)
Trigger	[TriggerObject].getNumObjects()
	[TriggerObject].getNumObjects(Object Index)
TriggerData	[TriggerData].enterTrigger(Trigger, ObjectId)
	[TriggerData].leaveTrigger(Trigger, ObjectId)
	[TriggerData].tickTrigger(Trigger)
WaterBlock	waterBlock.toggleWireFrame()
WorldEditor	worldEditor.redirectConsole(objID)

Table A.3 Torque Script Keywords

Keyword	Description
break	Breaks execution out of a loop.
case	Indicates a choice in a switch block.
continue	Causes execution to continue at top of loop.
datablock	Indicates that the following code block defines a datablock.
default	Indicates the choice to make in a switch block when no cases match.
do	Indicates start of a do-while type loop block.
`else`	Indicates alternative execution path in an if statement.
`false`	Evaluates to 0, the opposite of `true`.
for	Indicates start of a for loop.
function	Indicates that the following code block is a callable function.
if	Indicates start of a conditional (comparison) statement.
new	Creates a new object datablock.

continued

Team LRN

package	Indicates that the following code block encompasses a package.
return	Indicates return from a function.
switch	Indicates start of a switch selection block.
`true`	Evaluates to 1, the opposite of `false`.
while	Indicates the start of a while loop.

Table A.4 Torque Script Operators

Symbol	Meaning
+	Add.
-	Subtract.
*	Multiply.
/	Divide.
%	Modulus.
++	Increment by 1.
--	Decrement by 1.
+=	Addition totalizer.
-=	Subtraction totalizer.
*=	Multiplication totalizer.
/=	Division totalizer.
%=	Modulus totalizer.
@	String append.
( )	Parentheses—operator precedence promotion.
[ ]	Brackets—array index delimiters.
{ }	Braces—indicate start and end of code blocks.
SPC	Space append macro (same as @ " " @).
TAB	Tab append macro (same as @ "\t" @).
NL	New line append (same as @ "\n" @).
~	(Bitwise NOT) Flips the bits of its operand.
|	(Bitwise OR) Returns a 1 in a bit if bits of either operand is 1.
&	(Bitwise AND) Returns a 1 in each bit position if bits of both operands are 1s.
^	(Bitwise XOR) Returns a 1 in a bit position if bits of one but not both operands are 1.
<<	(Left-shift) Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in 0s from the right.
>>	(Sign-propagating right-shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off.
|=	Bitwise OR with result assigned to the first operand.
&=	Bitwise AND with result assigned to the first operand.
^=	Bitwise XOR with result assigned to the first operand.
<<=	Left-shift with result assigned to the first operand.

continued

Team LRN

>>=	Sign-propagating right-shift with result assigned to the first operand.
!	Evaluates the opposite of the value specified.
&&	Requires both values to be true for the result to be true.
||	Requires only one value to be true for the result to be true.
==	Left-hand value and right-hand value are equal.
!=	Left-hand value and right-hand value are not equal.
<	Left-hand value is less than right-hand value.
>	Left-hand value is greater than right-hand value.
<=	Left-hand value is less than or equal to right-hand value.
>=	Left-hand value is greater than or equal to right-hand value.
$=	Left-hand string is equal to right-hand string.
!$=	Left-hand string is not equal to right-hand string.
//	Comment operator—ignore all text from here to the end of the line.
;	Statement terminator.
.	Object/datablock method or property delimiter.

Table A.5 Torque Script Operator Precedence

High Priority						Low Priority
( )	*	/	%	+	-	=

Table A.6 Torque Script Tokens

Token	Description
string constant	A sequence of alphanumeric characters bracketed by single or double quotes.
variable	Prefixed with % for local variable or $ for global variable, which is then always followed by a letter character. After the initial letter character, there can be a series of alphanumeric characters, underscores, or colons; a variable cannot end with a colon.
identifier	An initial letter character followed by an optional sequence of alphanumeric characters or underscores.
number	A decimal integer or floating point number. Hexadecimal numbers can be used if the token begins with 0x (zero-x).

Team LRN

Table A.7 Torque Script String Formatting Codes

Code	Description
\r	Embeds a carriage return character.
\n	Embeds a new line character.
\t	Embeds a tab character.
\xhh	Embeds an ASCII character specified by the hex number (hh) that follows the x.
\c	Embeds a color code for strings that will be displayed on-screen.
\cr	Resets the display color to the default.
\cp	Pushes the current display color onto a stack.
\co	Pops the current display color off the stack.
\cn	Uses n as an index into color table defined by GUIControlProfile.fontColors.

Table A.8 Torque Datablocks

Datablock	Parent
AudioDescription	SimDataBlock
AudioEnvironment	SimDataBlock
AudioProfile	SimDataBlock
AudioSampleEnvironment	SimDataBlock
CameraData	ShapeBaseData
DebrisData	GameBaseData
DecalData	SimDataBlock
ExplosionData	GameBaseData
FlyingVehicleData	VehicleData
GameBaseData	SimDataBlock
HoverVehicleData	VehicleData
ItemData	ShapeBaseData
LightningData	GameBaseData
MissionMarkerData	ShapeBaseData
ParticleData	SimDataBlock
ParticleEmitterData	GameBaseData
ParticleEmitterNodeData	GameBaseData
PlayerData	ShapeBaseData
PrecipitationData	GameBaseData
ProjectileData	GameBaseData
ShapeBaseData	GameBaseData
ShapeBaseImageData	GameBaseData
SimDataBlock	*none*
SplashData	GameBaseData

continued

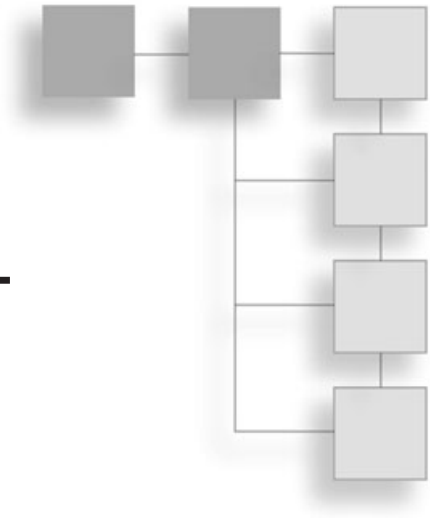
Team LRN

StaticShapeData	ShapeBaseData
TSShapeConstructor	SimDataBlock
TriggerData	GameBaseData
VehicleData	ShapeBaseData
WheeledVehicleData	VehicleData
WheeledVehicleSpring	SimDataBlock
WheeledVehicleTire	SimDataBlock

This page intentionally left blank

Team LRN

Appendix B



Game Development Resources on the Internet

Just about everything you could possibly need to know regarding game development can be found on the Internet. But then, you probably already knew this!

I've listed in this appendix every link from my own personal bookmark folder of game development resources, as well as a ton more recommended by friends.

Of course, this is certainly not an exhaustive list. However, the sites listed have very long histories on the Web, so most are not likely to succumb to "link rot."

I think you will find these resources useful.

Torque-Related Web Sites

ActionRPG

RPG-related enhancements for Torque.

http://www.actionrpg.com

BraveTree Productions

ThinkTanks home.

http://www.bravetree.com

Team LRN

GarageGames
Home of the Torque Engine.
http://www.garagegames.com

Gnometech
Torque modeling and other resources.
http://www.gnometech.com

Hall of Worlds
Torque development notes and tips.
http://www.hallofworlds.com

Holodeck: Virtual Reality Computing for Design
Torque and QuArK tutorials.
http://holodeck.st.usm.edu/vrcomputing/vrc_t

PlanetTribes—Torque
Source for Torque-related files and content.
http://www.planettribes.com/torque

Realm Wars Development Site
Cooperative game development project.
http://www.realmwarsgame.com

ToRK|Game|Dev|
Add-ons, code snippets, and tutorials.
http://tork.zenkel.com

Team LRN

Game Development Web Sites

3D Café

3D models and resources.

http://www.3dcafe.com

3D Today Magazine

3D modeling tutorials, resources, and articles.

http://3dtoday.com

3Dup.com

2D and 3D models and resources.

http://www.3dup.com

AngelCode

Game development and more.

http://www.angelcode.com

CFXweb

Game design, tutorials, and resources.

http://www.cfxweb.net

CodeGuru

Programming news, tutorials, and links.

http://www.codeguru.com

Dictionary of Algorithms and Data Structures

National Institute of Standards and Technology resource.

http://www.nist.gov/dads

Team LRN

Doctor Dobb's Journal
Programming news, articles, and links.
http://www.ddj.com

flipCode
Game development news and resources.
http://www.flipcode.com

Gamasutra
Game development news, articles, and resources.
http://www.gamasutra.com

Game Developer Magazine
Game development news, articles, and resources.
http://www.gdmag.com

Game Developers Conference
GDC news and promotional information.
http://www.gdconf.com

Game Developer's Lair
Game development news, articles, and resources.
http://www.gamedeveloper.net

GameDev.net
Game development news, articles, and resources.
http://www.gamedev.net

Team LRN

Game Institute
Professional training in the field of video game production and development.
http://www.gameinstitute.com

Gamer's Technical Resources
Game development news, articles, and resources
http://knockout.ww4.us:7979/index.php

iDevGames
Game development news, articles, and resources.
http://www.idevgames.com

insert credit
Gaming news, articles, reviews, and resources.
http://www.insertcredit.com

International Game Developers Association
Game development news, articles, and resources for independent developers.
http://www.igda.com

Linux Game Development Center
Game development news, articles, and resources.
http://lgdc.sunsite.dk

Linux Game Tome
Game development news, articles, and resources.
http://www.happypenguin.org

Team LRN

Machinima.com
Real-time 3D animation.
http://www.machinima.com

MathWorld
Math tutorials, articles, and resources.
http://mathworld.wolfram.com

Mesh Factory
Source for 3D models.
http://www.meshfactory.com

Monster Studios
Home of the Reaction Engine.
http://www.monsterstudios.com

NeHe Productions
Game technology articles and tutorials.
http://nehe.gamedev.net

NeXe
Game technology articles and tutorials.
http://nexe.gamedev.net

Oxford Dynamics
FastCar library—fast, precise, and simple library for vehicle simulation in games.
http://www.oxforddynamics.co.uk

Team LRN

Polycount
Game development articles, resources, and tutorials.
http://www.planetquake.com/polycount

Prefabland
Freeware 3D models source.
http://www.ejoop.com/pfl

Programmers Heaven
Programming articles, resources, and tutorials.
http://www.programmersheaven.com

Psionic 3D Design
3D modeling resources.
http://www.psionic3d.co.uk

SourceForge.net
Open Source software development Web site; large repository of Open Source code.
http://sourceforge.net

Steering Behaviors for Autonomous Characters
Paper by Craig Reynolds.
http://www.red3d.com/cwr/steer

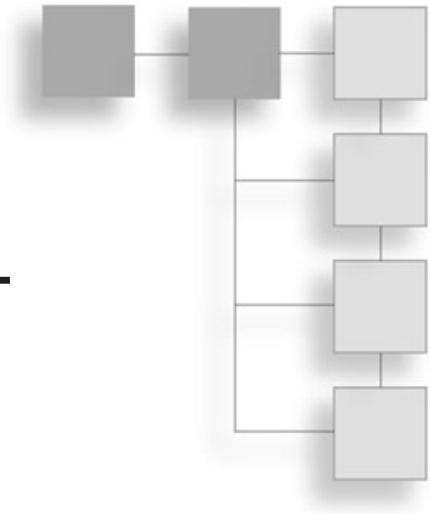
Wotsit's Format
Programming articles, resources, and tutorials.
http://www.wotsit.org

Team LRN

This page intentionally left blank

Team LRN

Appendix C



Game Development Tool Reference

All of the tools listed in this appendix are for Windows platforms. Some of the listed tools are available also for the Macintosh and Linux systems. For more information on Macintosh and Linux game development tools, see Table C.1 and Table C.2.

Table C.1 Linux Tool Sources on the Web

Site	Link
Linux Game Development Center	http://lgdc.sunsite.dk
Linux Game Tome	http://www.happypenguin.org
Tucows/Linux	http://download.tucows.com/perl/Linux.html

Table C.2 Macintosh Tool Sources on the Web

Site	Link
iDevGames	http://www.idevgames.com/
Mac's Heaven	http://www.mac-heaven.com/
Tucows/Macintosh	http://download.tucows.com/perl/Mac.html

Team LRN

Shareware and Freeware Tools

Modeling

Blender

3D modeling

Multiplatform: Windows, Linux, Irix, Sun Solaris, FreeBSD, or Mac OS X.

Free software: Open Source/GPL.

http://www.blender3d.org

gmax

3D modeling

Stripped-down version of 3ds max. Requires exporters to be useful.

Free. (Caveat: Few exporters are available; exporter developers must pay a very high fee.)

http://www.discreet.com/products/gmax

Hammer/Worldcraft

3D modeling—maps or levels

Worldcraft (later renamed Hammer) was written for creating Half-Life maps.

Free to be used only for creating Half-Life levels or for use by developers using the Torque Engine. Plug-ins available for Torque .DIF format. Windows only.

http://collective.valve-erc.com/

MilkShape 3D

3D modeling

Supports Torque using exporter plug-in. Windows only.

INCLUDED ON COMPANION CD

http://www.swissquake.ch/chumbalum-soft

QuArK

3D modeling—maps or levels

Originally written for creating Quake maps (QUake ARmy Knife). Supports Torque .DIF format. Windows only.

INCLUDED ON COMPANION CD

http://dynamic.gamespy.com/~quark

Image Editing

Paint Shop Pro

Image editing

Fully featured image processing, painting, and editing tool.

INCLUDED ON COMPANION CD

http://www.jasc.com

Programming Editing

Tribal IDE

Text editing and debugging

Integrated debugger-editor written specifically to work with Tribes 2 and Torque. Useful to have around for debugging.

http://depot.gamerzcore.com/

UltraEdit-32

Text editing

Includes project and workspace features as well as macros.

INCLUDED ON COMPANION CD

http://www.ultraedit.com

Audio Editing

SoundEdit Pro

Audio editing and sound processing

Allows manual editing of sound files and conversion between many types.

$39.95

http://www.rmbsoft.com/sep.asp

Audacity

Audio editing and sound processing

Allows manual editing of sound files, recording, and wave manipulation.

INCLUDED ON COMPANION CD

http://audacity.sourceforge.net/

UVMapper

3D UV texture mapping utility

Allows user to completely remap the model textures of a Wavefront (obj) model.

INCLUDED ON COMPANION CD

http://www.uvmapper.com

Retail Tools

3ds max

3D modeling

Popular commercial 3D modeling software for Windows.

$3,000 (Price is approximate—may vary according to reseller and discount eligibility.)

http://www.discreet.com

Adobe Photoshop

Image editing

Popular fully featured image processing, painting, and editing tool.

$649

http://www.adobe.com

Corel Painter

Image editing

Popular commercial paint program for Windows.

$299 (Price varies—sometimes lower with special offers.)

http://www.corel.com

Deep Paint 3D

Image editing

Popular commercial paint program for Windows.

$995

http://www.righthemisphere.com

Deep UV

3D UV texture-mapping utility

Fully featured commercial product targeted to professionals.

$649.99

http://www.righthemisphere.com

Maya

3D modeling

Popular commercial 3D modeling software for Windows.

$1,999 to $6,999 (Price depends on product set.)

http://www.alias.com

Team LRN

Poser

3D animation editing

Fully featured commercial product with rendering and automated tools.

$249

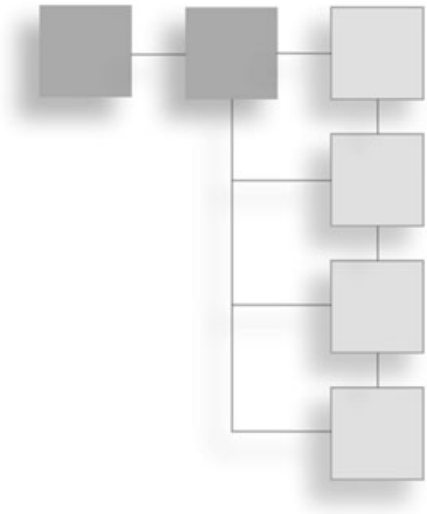
http://www.curiouslabs.com

GNU General Public License

The software packages Audacity and QuArK, listed in this appendix and included on the companion CD, are distributed under the terms of the GNU General Public License (GPL).

Please review the license agreement on the CD before downloading and using the software.

Appendix D



QuArK Reference

Chapter 17 described how to use QuArK to create some structures. This appendix contains reference material for using QuArK, as well as details about how to use map2dif, the utility with Torque that's used to generate Torque DIF files.

The Map Editor

When you launch QuArK, you immediately get the QuArK Explorer window. This is your jumping-off point from which you can get to the various main features of the program. The most important feature of QuArK is the Map Editor, which you have already encountered, shown in Figure D.1.

tip

Whenever you see the text "Press F1 for help" in the *hintbox* at bottom left of the window, you can press F1 and get a larger hintbox that gives some information about the item under the cursor.

These hints are especially useful if you are new to QuArK and you have forgotten the function of some visible feature in the window. Press F1 to get a quick hint to refresh your memory.

Move your mouse out of the hintbox to make it disappear.

Remember our previous sidebar discussion of structures vs. interiors and maps vs. rooms in Chapter 17? After all the terminological torture, it boils down to either a part of, or the entire, 3D world in which you play your games. So in the context of developing our game for Torque, we use QuArK to make structures or interiors. In the context of QuArK as a non-specific development tool, we make maps or rooms. The difference lies in how you use them.

Team LRN

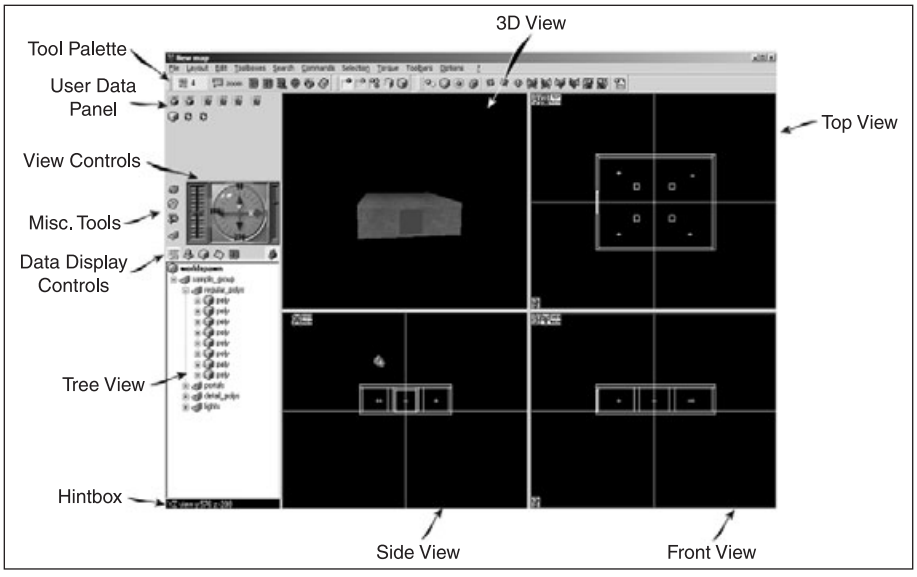


Figure D.1 The Map Editor.

The remainder of this chapter is about how to use many of the features of QuArK. We will stick to the QuArK terminology so that the words here match the words in the program.

A map is made out of mathematical *polyhedrons* in 3D space. As mentioned earlier, these are also called brushes. Brushes will present in your game as a wall, floor, ceiling, furniture, or anything with a fixed collidable 3D shape. Weapons, lights, and other entities are not made out of brushes, and you cannot use the Map Editor to build their shape.

note

A polyhedron is a solid figure bounded by plane polygons or faces. Cubes and pyramids are specific variants of the polyhedron.

QuArK has a formidable array of features, and an equally formidable menu bar. In this section we can make them a bit less formidable with tables for each menu describing their functions.

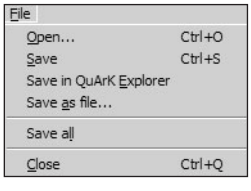


Figure D.2 File menu.

The File menu, as shown in Figure D.2, is pretty much your standard File menu with some QuArK-specific features. Table D.1 contains an itemized description of the menu.

Table D.1 File Menu

Menu Item	Description
Open	Opens any QuArK-recognized file.
Save	Saves your current project to a .qrk file.
Save in QuArK Explorer	Use this if you want to add the current map to your current project. You can have multiple revisions of a single map contained in a single project. It's a good idea to give them distinctive and meaningful names so that you can tell them apart. The Map Editor will automatically close itself when you use this function.
Save as file	Use this to export just the current map to a .map or .qkm file.
Save all	Saves all opened files, including the current project and any modified add-ons.
Close	Closes the Map Editor.

The Layout menu is used to manage the visual appearance of the QuArK Map Editor. It is depicted in Figure D.3, and Table D.2 contains an itemized description of the menu.

The Edit menu is used to provide map-editing features. It is depicted in Figure D.4, and Table D.3 contains an itemized description of the menu.

The Toolboxes menu is used to provide access to various tools available in the Map Editor. It is depicted in Figure D.5, and Table D.4 contains an itemized description of the menu.

The Search menu is used to locate various parts of a map in the Map Editor. It is depicted in Figure D.6, and Table D.5 contains an itemized description of the menu.

The Commands menu is used to provide access to various functions in the QuArK Map Editor. It is depicted in Figure D.7, and Table D.6 contains an itemized description of the menu.

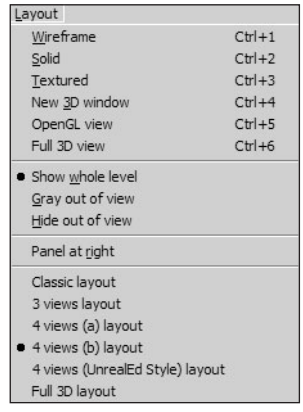


Figure D.3 Layout menu.

Figure D.4 Edit menu.

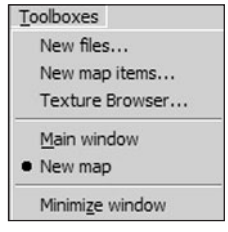


Figure D.5 Toolboxes menu.

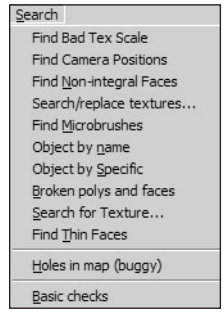


Figure D.6 Search menu.

Table D.2 Layout Menu

Menu Item	Description
Wireframe	Changes all map-views to show polyhedrons and entities as wireframe graphics. This is the fastest drawing method of the map views.
Solid	Changes all map views to show polyhedrons as solid objects. This means that you won't be able to "see-through" polyhedrons once they are in view.
Textured	Changes all map views to show polyhedrons with their applied textures and alignment. It will also show 3D models of entities, if QuArK can find the 3D representation.
New 3D window	Creates a new free-floating 3D edit window. An "eye" with an angle handle will be shown in the map views to illustrate from where the camera sees the world. You can actually edit polyhedrons in the 3D window as well as manipulate textures.
OpenGL view	Creates a new free-floating 3D edit window using the OpenGL standard 3D graphic library. In QuArK v5.10 it is also the only way to preview (colored) light effects. (Silicon Graphics' OpenGL drivers for Windows: http://www.berkelium.com/OpenGL/sgi-opengl.html.)
Show whole level	Draws, in wireframe mode, all lines in the map, even those that can't be seen simultaneously in all map views.
Gray out of view	Draws, in wireframe mode, gray lines for those parts that can't be seen within the map views.
Hide out of view	Draws, in wireframe mode, only those lines that are visible in the field-of-view.
Panel at right	When checked, compass and data views are presented on the right-hand side of the display.
Layouts	Use this menu item to switch between the different layouts available.

Table D.3 Edit Menu

Menu Item	Description
Undo *action*	This function will be active if it is possible to undo a previous action. The *action* is what it will undo.
Undo/Redo	Open up the Undo/Redo window and you can undo or redo more actions with one mouse click.
Cut	Remove the selected object and put it into the clipboard.
Copy	Copy the selected object into the clipboard without removing it from the map.
Paste	Paste the data in the clipboard to where you indicate in the map, as long as it is data that QuArK understands.
Duplicate	Combines the Copy and Paste functions, without leaving a copy in the Clipboard.
Delete	Deletes the selection. Nothing is put in the Clipboard.
View group	For adjusting the view properties of groups. Only active when you've selected a group in the tree view.

Team LRN

Table D.4 Toolboxes Menu

Menu Item	Description
New files	Shows the new files toolbox.
New map items	Shows the new map items toolbox.
Texture Browser	Shows the texture browser toolbox.
Main window	Shows the main QuArK Explorer window.
New map	Refers to the maps that you are currently editing. There may be one or more entries here. If you have more than one map in your current project, they will all be listed here, and the one that is current will be marked with a bullet beside it. If your map is not named, this entry will be listed as New map, otherwise it will be the name of the map.

Table D.5 Search Menu

Menu Item	Description
Replace textures	Used to replace one texture in a selection with another specified texture.
Entities by name	Search for the next entity that has a class name that matches the specified name.
Entities by specific	Search for the next entity with a specific name that matches the one specified.
Broken polys and faces	Search the map for invalid polyhedrons and faces that do not belong to any brushes in the map.
Holes in map	Check your map for leaks. A leak would be an area of the map that is exposed to the "outside" world. This function really only applies to maps that are of interiors, and only really matters if you are using internal lights with portals to block doorways and windows.
Basic checks	Performs some basic sanity checks on the map to ensure that they will compile properly.

note

Brush subtraction is a method of making a hole in a brush by using another brush to provide the shape of the hole.

First, create a brush with the shape of the hole and place it where you want the hole to appear; this means that your "hole brush" will end up overlapping at one or more other brushes. The overlapped brush or brushes will have the their shapes modified with the volume of the "hole brush" turned into empty space. You then must delete the "hole brush" for the hole to appear.

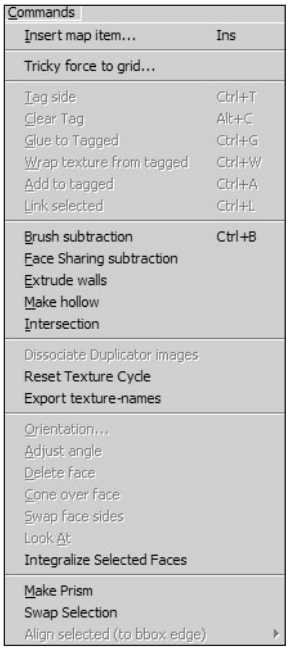


Figure D.7 Commands menu.

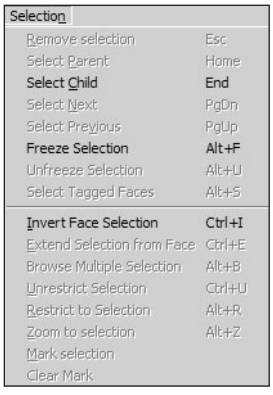


Figure D.8 Selection menu.

The Selection menu is used to provide methods for selecting portions of a map in the QuArK Map Editor. It is depicted in Figure D.8, and Table D.7 contains an itemized description of the menu.

The Torque menu is used to provide access to functions that are specific to the Torque map2dif exporting tool via the Map Editor. It is depicted in Figure D.9, and Table D.8 contains an itemized description of the menu.

The Toolbars menu is used to provide access to functions that manage the toolbars in the QuArK Map Editor. It is depicted in Figure D.10, and Table D.9 contains an itemized description of the menu.

The Options menu is used to provide a means to adjust various options in the Map Editor. It is depicted in Figure D.11, and Table D.10 contains an itemized description of the menu.

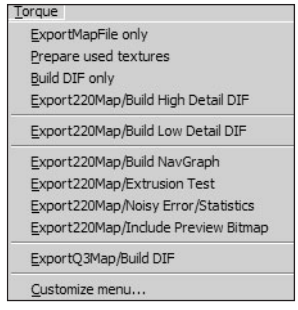


Figure D.9 Torque menu.

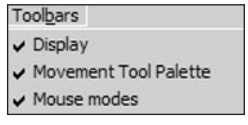


Figure D.10 Toolbars menu.

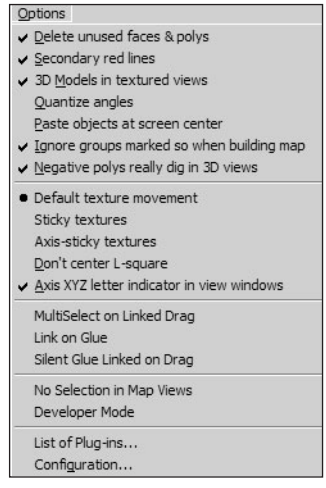


Figure D.11 Options menu.

Table D.6 Commands Menu

Menu Item	Description.
Insert map item	Opens the New map items window and inserts chosen item.
Tricky force to grid	Repairs maps that protrude off the grid.
Tag side	The functionality for this comes from a plug in. With it, you can tag a face of a polyhedron. The following three functions require this function to be operational.
Clear tag	Removes a tag.
Glue to tagged	Glues another selected brush to the tagged face.
Wrap texture from tagged	Enabled when the selected face is next to the tagged face, and will seamlessly wrap the texture from the tagged face onto the selected face.
Add to tagged	Enabled when there is a tagged face and another face is selected; adds the selected face to the collection of tagged ones.
Link selected	Links the selected faces so that if one is moved, you will be able to move the other linked faces along with it.
Brush subtraction	Use this command to subtract one brush from another brush by first selecting the brush you want the subtraction to occur on and then selecting the brush that should be subtracted from the first, and then choosing this function.
Face sharing subtraction	An advanced brush subtraction function. If a face from the original brush will be split up into multiple faces by a brush subtraction, using this function will cause that face to split in a way that allows it to be shared by the other brushes created by the brush subtraction.
Extrude walls	Extrudes walls from all the faces of the selected brush or brushes and deletes the originals.
Make hollow	Makes brush for each face the selection has and deletes the original. The inflate/deflate value in the Movement configuration window dictates the thickness of the new brushes and whether they should occupy or surround the original selected brushes.
Intersection	Adds two brushes together; attempts to create a new brush which occupies the common area of the selected brushes.
Dissociate duplicator images	Active if you have marked a duplicator. Creates actual copies of the duplicator object or objects then removes the duplicator.
Reset Texture Cycle	Reloads the files that specify texture cycles for duplicators.
Export texture names	Exports a list of textures used by this map to a text file.
Orientation	Active if you have a face selected. Brings up a window in which you can edit the face's compass angle and inclination.
Adjust angle	Active if you have a face selected. Adjusts the angle of a face to be rounded to the nearest angle that is a multiple of that specified in the Force Angle property in the Map, Building configuration window.
Delete face	Deletes a face. Note that deleting a face from a brush will probably invalidate the brush.

continued

Team LRN

Cone over face	Creates a new set of faces shaped like a cone on top of the selected face. The number of edges of the selected face will dictate the number of new faces created.
Swap face sides	Inverts the normals of the selected faces.
Look At	Forces the 3D view to center on the selected face so that you can view it from a right angle to the face's plane. Hold down the Shift key to look at it from the rear.
Integralize Selected Faces	Adjusts faces that are lacking sufficient coordinates with integral vertices. Changed faces are presented to the user for verification.
Make Prism	Creates a prism of various types.
Swap Selection	Swaps contents of two selected items.
Align selected	Aligns items in selection along their bounding box edges, or along the edges of a marked object

Table D.7 Selection Menu

Menu Item	Description
Remove Selections	Unselects all objects that are currently selected.
Select parent	Changes selection to that of the currently selected item's parent. For example, if you have a face selected, the selection will change to that face's parent brush.
Select child	Changes selection to the first child of currently selected item.
Select next/previous	Changes selection to the next or previous child of currently selected item's parent.
Freeze selection	Helps prevent accidentally deselecting an item by clicking in the wrong place. When this is active, you need to hold down the left-shift key to change a selection.
Unfreeze selection	Disables the selection freeze function.
Select tagged faces(s)	If you have more than one tagged face, this is how you can select them all at once.
Invert face selection	Changes the selection to be the currently non-selected faces in the same brush or brushes as the current selection.
Extend selection from face	Extends the selection to all co-planar adjacent faces of neighboring brushes.
Browse multiple selection	Presents a dialog that permits you to choose individual elements of a multiple selection to select and work on as a sub-set.
Unrestrict selection	Disables the Restrict to Selection function.
Restrict to selection	Restricts the Map Editor to work only on the selected items. Other material will be hidden or grayed out depending on the gray out/hide out of view setting.
Zoom to selection	Zooms the map view to be filled with the selection.
Mark selection	Marks for processing with the Reorganize Tree function.
Clear Mark	Clears any Mark selection marks

Team LRN

Table D.8 Torque Menu

Menu Item	Description
ExportMapFile only	Saves the current map in .map format.
Prepare used textures	Scans the current map and copies all textures used into the directory specified in the Games, Torque configuration window
Build DIF only	Runs map2dif and generates the .dif output file without saving the map or gathering the textures.
Export220Map/Build High Detail DIF	This is an all-in-one function: it saves the map file, performs an exhaustive BSP-search on the map before generating the high-detail .DIF output, and gathers the used textures. In this book, the map output is deposited in the directory: C:\QuArK 6.3\torque\tmpQuArK\maps and the textures are copied to: C:\QuArK 6.3\torque\tmpQuArK\textures.
Export220Map/Build Low Detail DIF	This is an all-in-one function: It saves the map file, performs a minimal and quick BSP-search on the map before generating the low-detail .DIF output, and gathers the used textures.
Export220Map/Build NavGraph	Generates navigation graph information that shows the relationship between faces, brushes, and entities in the map. Also gathers textures.
Export220Map/Extrusion Test	Generates .dif output and verifies extruded placement of brushes. Also gathers textures.
Export220Map/Noisy Error/Statistics	Generates .dif output and provides detailed statistics and error data for any problems that are encountered. Also gathers textures.
Export220Map/Include Preview Bitmaps	Generates .dif output and creates a preview bitmap for inclusion in the .dif file. Also gathers textures.
Export220Map/Build DIF	Generates the .dif output file from a Quake 1 or Quake 2 map file. Gathers textures as well.
Customize menu	Allows you to add, remove, or otherwise modify the contents of the Torque menu.

Table D.9 Toolbars Menu

Menu Item	Description
Display	Toggles Display toolbar visiblilty.
Movement tool palette	Toggles Movement toolbar visiblilty.
Mouse modes	Toggles Mouse Modes toolbar visiblilty.

Team LRN

Table D.10 Options Menu

Menu Item	Description
Delete unused faces & polys	Prevents creation of invalid brushes and faces when checked.
Secondary red lines	Causes an extra set of red lines to appear in the map views. Use this to reduce the selection area of one map view if you do the selection in the other.
3D models in textured views	Causes the textured views to be rendered as 3D models.
Adjust angles automatically	When this is checked, the program will automatically adjust the angle of the selected item to be rounded to the nearest angle that is a multiple of that specified in the Force Angle property in the Map, Building configuration window.
Paste objects at screen center	Causes pasted items to appear at screen center rather than at the original location.
Ignore groups marked so when building	Any groups marked Ignore will be ignored when the map is compiled.
List of plug-ins	Shows what plug-ins QuArK has loaded.
Configuration	Launches the Configuration utility.

Configuration Utility

The following sections contain many of the more important configuration parameters and their meanings.

General

Display

Gamma correction. Use this to adjust the brightness of texture colors.

Window captions. Sets whether to color window captions.

Activate windows on. When enabled, windows are auto-activated when the mouse cursor is inside them.

If not selected and **If selected.** Determines whether selected icons should be displayed in blue or brown.

3D View

Select 3D viewer. If you have a 3Dfx-compatible 3D acceleration graphics card installed, QuArK will use it to speed up drawing in 3D windows. If you don't have one, you must select Software only.

Entities in 3D. Whether or not QuArK should draw entities (3D models) in the 3D windows.

Team LRN

"Far" distance. How deep QuArK should draw in the 3D windows. Lower values will speed up drawing.

Vertical view angle. The field of view of the perspective views, in degrees. Just as with cameras, you enter the height of the view not the width.

Frame color. This is the color of the bounding area of the 3D windows, outside the area where QuArK renders.

3Dfx card owners only

These settings apply to those users that have 3dfx videos cards.

Software drivers only.

If you have selected Software only in the 3D viewer, then here is where you can adjust how fast the software renderer should be able to draw things in the 3D window, while standing still and while walking/moving around.

Mouse sensitivity

These values control the mouse sensitivity when using a mouse to move the camera in the 3D views.

Keyboard settings

Instead of using the mouse to move around with (which can be quite difficult at times), you can either use these standard movement keys when the 3D window is active, or redefine them to suit your needs. It is highly recommended that you know how to move around in 3D window using the keys.

Keyboard movement dynamics

Change these values if you feel that moving around using the keys isn't quite fast or slow enough. You must experiment a bit to find a setting that you like using.

OpenGL

Settings in this folder are only usable if you have a true OpenGL graphics card.

3D accelerators like 3Dfx, Voodoo, Voodoo-II, and other low-priced cards are not truly OpenGL-compatible—they only emulate it through a glide driver.

Memory

Values in this settings folder control how much memory QuArK is allowed to consume and the number of undo levels it should store.

Map

Mouse

To get the most out of the Map Editor, you can specify here how the mouse should act when you do certain operations to it, like pressing a button and dragging.

It is highly recommended that you familiarize yourself with these mouse settings or redefine them if you like. But you should experiment with them in the Map Editor to get a feel of what they do.

You should be aware, however, that the Mouse Modes Tool palette sometimes will overrule the settings you've chosen in this setup folder.

Keys

Some functions in the Map Editor map views can also be accomplished by pressing keys. In this settings folder, you can see and redefine keys for the actions shown here.

Options

A lot of things in the Map Editor are controlled in this folder. A selection will be described here, but you should experiment with the settings yourself to get a knowledge of what they do and to see if you can use them for something useful.

Draw axis. Indicates whether the axis bug (the X-, Y-, and Z-axes through coordinate (0,0,0)) should be drawn.

Show indirect 'target' links. Shows with colored arrows what a selected entity has as targets, and their targets, and their targets, and their targets, and their targets, and so on. Having this on can really slow down map drawing in the Map Editor, if you've got a lot of targets. It is recommended to only have this selected when you want to see that it's all connected.

Both red lines. Turn on a second set of red guidance lines.

Cross-like cursor. If you're doing precision work, it's better to have a cross instead of the arrow-shaped mouse cursor.

Auto adjust normal. When this box is selected, face normals are forced to the nearest multiple of 15 degrees when they are adjusted (15 is the default value and can be changed). This works just as if you were holding down the Ctrl key all the time.

Delete unused faces. Unless you know what you're doing, keep this selected at all times.

Dig in 3D views. To see what a digger or negative polyhedron will do to your level in the 3D windows, keep this selected.

Hide faces in tree. Select this if you do not want to be able to expand polyhedrons showing their faces in the Tree view. If you are working with shared faces, clear this.

Team LRN

Hint for map handles. Keep this selected or you will not be able to get the great flyover hint boxes on handles.

Don't write //TX# in .map. If you do not use build tools that are custom-made for QuArK, you won't be able to take advantage of the better texture alignment when compiling your map. This option must be cleared when you use TXQBSP/TXQCSG build tools.

Building

This section controls some default values that will affect your map making.

Def. brush with entities. Selects the size of a default cube you want when creating a new entity.

Wall width (for Make Hollow). The default thickness in units that the make hollow action should result in.

Force angle to (degrees). The closest minimum degree angle that a rotation should snap to.

map2dif Reference

The tool we use in QuArK for compiling maps to create .DIF structures or interiors is, as we have already seen, map2dif.exe. This program is part of the Torque SDK package. The version of map2dif that I have included for use with QuArK is map2dif_DEBUG, which provides extra diagnostic information that can help solve map error problems.

The tool can be used outside of QuArK by invoking it from the command shell. The syntax for using map2dif is as follows:

```
map2dif [-v][-p][-s][-l][-h][-g][-e][-n][-o outputDirectory][-t textureDirectory]
file.map
```

Switches:

- `-v` Print program version information
- `-p` Include a preview bitmap in the interior file
- `-d` Process only the detail specified on the command line
- `-l` Process as a low detail shape (implies -d)
- `-h` Process for final build (exhaustive BSP search)
- `-g` Generate navigation graph info
- `-e` Do extrusion test

`-n`	Noisy error/statistic reporting
`-s`	Don't search for textures in parent directory
`-q <1,2>`	Parse Quake map file, version 1 or 2
`-o dir`	Directory in which to place the .dif file
`-t dir`	Location of textures
`file.map`	Name of file to be processed

The program takes a map file, processes it according to the supplied switches, and produces as output a Torque .DIF file. The .DIF file is deposited in the same directory as the .MAP file unless the -o switch is employed to specify the output directory.

The textures used in the map have to be loaded in order to process the map. The map2dif tool needs to load the textures so that it can determine the width and height of each texture. It uses this information to calculate polygon texture mapping information that will be included in the .DIF file. If you change the size of a texture, you may need to re-process any map files that use that texture.

When it runs, map2dif looks in the default directory of the .MAP input file; it then recursively looks in its parent directory and on upwards until it reaches the root directory, or finds the texture in question. If the -t *dir* switch is used in the command line, the program starts searching for the textures in the directory specified by dir. The program will load either .JPG, or .PNG forms of the textures specified in the map.

When it runs, the Torque engine expects textures to be in the same directory as the map files that it uses, or in a parent directory. Parent directories are searched all the way to the root main directory (where the torque engine executable resides). The root directory itself is never searched.

The map2dif tool uses certain texture names to help identify special brushes. These textures are never loaded but they need to be in the base texture directory of QuArK to allow processing of your maps.

- null.png
- origin.png
- trigger.png
- forcefield.png

Table 17.11 contains a list of the entities supported by map2dif and the various entity attribute options.

Table 17.11 map2dif Supported Entities

Category	Entity	Attribute	Data Type
Core Entities			
	worldspawn		
	detail_number	int (default is 0)	
	min_pixels	int (default is 250)	
	geometry_scale	int Must be a power of 2 (default is 32)	
	light_geometry_scale	int Must be a power of 2 (default is 32)	
		ambient_color	color (default is 0,0,0)
		emergency_ambient_color	color (default is 0,0,0)
	detail		
	collision		
	vehicle_collision		
	portal		
	ambient_light	bool: Pass ambient light (default is 0)	
	target		
		name	string (default "")
		origin	pos (default 0,0,0)
Light Emitters			
	light_emitter_point		
		origin	pos (default 0,0,0)
		target	
		state_index	
		falloff_type	bool: 0 = distance, 1 = linear (default false)
		falloff1	float (default is 10)
		falloff2	float (default is 100)
		falloff3	float (default is 0)
	light_emitter_spot		
	origin	pos (default is 0,0,0)	
		target	
		state_index	
		falloff_type	bool: 0=distance, 1=linear (default false)
		falloff1	int (default is 10)
		falloff2	int (default is 100)
		falloff3	int (default is 0)

continued

Team LRN

		direction	vector (default is 0,0,-1)
		theta	radian: inner angle (default is 0.2)
		phi	radian: outer angle (default is 0.4)
Lights			
	light		
		name	string (default "")
		origin	pos (default is 0,0,0)
		spawnflags	int: animation flags...
		alarm_type	bool (default is 0)
		state	int: State number
		duration	float: State duration
		color	color: State color
Scripted Lights			
	light_omni		
		name	string (default "")
		origin	pos (default is 0,0,0)
		color	color (default is 1,1,1)
		alarm_type	bool (default is 0)
		falloff1	int (default is 10)
		falloff2	int (default is 100)
	light_spot		
		name	string (default "")
	origin	pos (default is 0,0,0)	
		target	
		color	color (default is 1,1,1)
		alarm_type	bool (default is 0)
		falloff1	int (default is 10)
		falloff2	int (default is 100)
		distance1	int: Inner distance (default is 10)
		distance2	int: Outer distance (default is 100)
Animated Lights			
	light_strobe		
		name	string (default "")
		origin	pos (default is 0,0,0)
		target	
		spawnflags	int: animation flags...
		color1	color (default is 0,0,0)

continued

	color2	color (default is 1,1,1)
	alarm_type	bool (default is 0)
	falloff1	int (default is 10)
	falloff2	int (default is 100)
	speed	int (default is ?)
light_pulse		
	name	string (default "")
	origin	pos (default is 0,0,0)
	spawnflags	int: animation flags...
	color1	color (default is 0,0,0)
	color2	color (default is 1,1,1)
	alarm_type	bool (default is 0)
	falloff1	int (default is 10)
	falloff2	int (default is 100)
	speed	int (default is ?)
light_pulse2		
	name	string (default "")
	origin	pos (default is 0,0,0)
	spawnflags	int: animation flags...
	color1	color (default is 0,0,0)
	color2	color (default is 1,1,1)
	alarm_type	bool (default is 0)
	falloff1	int (default is 10)
	falloff2	int (default is 100)
	attack	float (default is 1)
	sustain1	float (default is 1)
	sustain2	float (default is 1)
	decay	float (default is 1)
light_flicker		
	name	string (default "")
	origin	pos (default is 0,0,0)
	spawnflags	int: animation flags...
	color1	color (default is 1,1,1)
	color2	color (default is 0,0,0)
	color3	color (default is 0,0,0)
	color4	color (default is 0,0,0)
	color5	color (default is 0,0,0)
	alarm_type	bool (default is 0)
	falloff1	int (default is 10)
	falloff2	int (default is 100)
	speed	int (default is ?)

continued

Team LRN

	light_runway		
		name	string (default " ")
		origin	pos (default is 0,0,0)
		spawnflags	int: animation flags...
		color	color (default is 1,1,1)
	target		
		alarm_type	bool (default is 0)
		falloff1	int (default is 10)
		falloff2	int (default is 100)
		speed	int (default is ?)
		steps	int (default is 0)
		ingpong	bool (default is 0)
Special Entities			
	mirror_surface		
		origin	pos (default is 0,0,0)
		alpha_level	int (default is ?)
	door_elevator		
		name	string (default is " ")
		path_name	string (default ?)
		trigger[0-7]_name	string (default ?)
	force_field		
		name	string (default " ")
		color	color (default is 0.5,.8,1)
		trigger[0-7]_name	string (default ?)
	ai_special_node		
		name	string (default " ")
		origin	pos (default is 0,0,0)
Path Entities			
	path_node		
		name	string (default " ")
		next_node	string (default ?)
		next_time	int (default ?)
	path_start		
		name	string (default " ")
		next_node	string (default ?)
		next_time	int (default ?)
Trigger Entities			
	trigger		
		name	string (default " ")

Team LRN

INDEX

Symbols

+ (addition operator), 61
& (ampersand), 73
*** (asterisk),** 56, 61
$ (dollar sign), 54
// (double-slash operator), 50
! (exclamation point), 73
/ (forward slash), 61
*** (multiplication operator),** 62
\n (newline character), 124
% (percent sign), 54
| (pipe operator), 61
+ (plus sign), 56
\r (return character), 124
; (semicolon), 52
- (subtraction) operator, 61
\t (tab character), 124
~ (tilde), 39, 231
_ (underscore), 53

Numerics

1st PPOV (First-Person Point-of-View) games, 3
3D Cafe Web site, 743
3D objects
coordinate systems, 90–91
overview, 89
shapes, 94–95
transformation, 95–98
3D sounds, 559
3D Today Magazine Web site, 743
3Dup.com Web site, 743
3rd PPOV (Third-Person Point-of-View) games, 3

A

About UVMapper command (UVMapper Help menu), 408
action games, 3
action maps, console commands and, 24
ActionMap object, 174
ActionRPG Web site, 741
actions, lag, 133
activateDirectInput() function, 667
activateKeyboard() function, 667
activatePackage() function, 667
Add Dirt function, 537
Add flag (special materials, MilkShape), 462
Add Label At Selection command (Audacity Project menu), 548
Add Noise dialog box (Paint Shop Pro), 286
add-ons, 132
AddCardProfile() function, 668
addition operator (+), 61
AddMaterialMapping() function, 668
addMessageCallback function, 268
AddOsCardProfile() function, 669
AddTaggedString() function, 669
AddToServerGuidList function, 241
Adjust Height function, 537
Adjust Selection function, 537
adjustment layers, 292
administrative tools, as support infrastructure, 22
adventure games, 3–4
AI (Artificial Intelligence), 322
AiConnect() function, 669
Airbrush tool (Paint Shop Pro), 296–297, 517
alGetString() function, 669

Align Tracks Together command (Audacity Project menu), 548
Align with Zero command (Audacity Project menu), 548
alignment
cylindrical mapping option, 412
planar mapping option, 411
spherical unwrapping option, 413
alListener3f() function, 670
AllowConnections() function, 670
alpha channels, 288–290
alpha test phase, 661
Alphamap Browse Button button (MilkShape Materials tab), 402
AlPlayer class, 728
altCommand property, 642
alxCreateSource() function, 670
alxGetChannelVolume() function, 671
alxGetListener3f() function, 671
alxGetListenerf() function, 671
alxGetListeneri() function, 671
alxGetSource3f() function, 672
alxGetSourcef() function, 672
alxGetSourcei() function, 672
alxListener() function, 673
alxListenerf function(), 552
alxIsPlaying() function, 673
alxPlay() function, 552, 673
alxSetChannelVolume() function, 673
alxSource3f() function, 674
alxSourcef() function, 674
alxSourcei() function, 675
alxStop() function, 675
alxStopAll() function, 675
Ambient button (MilkShape Materials tab), 402
ampersand (&), 73
Amplify command (Audacity Effect menu), 549
AND operator, logical expressions, 73
AngelCode Web site, 743
Animate button (Keyframer tool), 404
Animate menu (MilkShape 3D), 394
animateTexture property, 601
animation
blended, 445
character modeling
joint animation, bone movement during, 445
rigging, 444
skeletal animation, 446–458
torque-supported sequences, 444
death, 455–457
look, 455
material, 27
mesh, 27
node/bone, 27
program example, 115–117
schedule() function, 115
sequences, 463
Torque Game Engine, 27
triggers, 209–210
vehicle model
body of vehicle, 467–472
collision mesh, 476
fenders, 473–474
mount nodes, 475–476
sketches, 466–467
skins, 476
testing, 477–478
wheels, 476–477
visibility tracks, 27
Animation FPS option (Preferences dialog box), 405
Animation Settings option (Torque Game Engine (DTS) Exporter dialog box), 460
animTexName property, 593
anti-aliasing, text, 306
APIs (Application Programming Interfaces), 10
arbitrary extrusion, character models, 417
area triggers, 209
arguments. *See* parameters
arithmetic operators, 63
arms, character models, 433–438
arrays
defined, 56
elements and, 56
Fruit program example, 56–58
index of, 56
subscripts and, 59
arrow widgets, 336
Artificial Intelligence (AI), 322
artwork, textures, 357–358
assemblers, defined, 46
assembly language, defined, 46
Assign button (MilkShape)
Joints tab, 403
Materials tab, 402

Assign command (UVMapper Edit menu), 408
assigned statements, 52
assignment strings, 59
asterisk (*), 56, 61
atmospheric perspective, 520
Attach & Defend, 3
Audacity tool
commands, shortcut keys to, 550
installing, 540
main screen, 542–543
menus
Edit menu, 545, 547
Effect, 545, 549
File, 545–546
Project, 545, 548
View menu, 545, 547
Play button, 541
Record button, 540–541
sample rates, 544
toolbar tools, 543
Track Panel tools, 544
Track Types tool, 544
volume control, 541
audio. *See also* sound
audio profiles, 121
editing tools, 751
program example, 119–121
Audio tool (Audacity tool), 544
AudioDescription property, 561
AudioProfile property, 561
Auto Proof button (Paint Shop Pro), 280
Auto Save option (Preferences dialog box), 406
Auto Tool button (MilkShape Model tab), 400
auto-update programs, as support infrastructure, 22
avatar inhabits, 3
axial extrusion, character models, 416
axis, coordinate systems, 90

B

back animation sequence, torque-supported, 444
backfaces, 95
background layers, 292
Backtrace() function, 675
BackUp function, 175
bandwidth strategies, 28
BanList class, 728
bar widgets, 336
barriers, 607
BassBoost command (Audacity Effect menu), 549
beta test phase, 661
Billboard flag (special materials, MilkShape), 462
billboard trees, creating, 488–490
BillboardZ flag (special materials, MilkShape), 462
binary systems, 46
Bitmap function, 537
bitmaps
bitmap images, 288–289
bitmap property, 164
chunked, 337–338
Blackhawk Down, 22
Blend function, 537
blended animation, 445
Blender class, 26
blur, terrain, 374
body of vehicle models, 467–472
BodyFriction command, 618
BodyRestitution command, 618
bone movement, during joint animation, 445
bookmark capabilities, UltraEdit-32, 41
Boolean logic, 72
bottom property, 164
bounded loops, 64
Box button (MilkShape Model tab), 399
Box Mapping dialog box, 409
Box tool (MilkShape), 479–480
Box unwrapping method, 409, 411
BrakeTorque command, 618
branches, tree creation, 486–487
branching, 74–75
break keyword, 54, 735
brick textures, 359
bridge structures, 505–508
brushes (Torque Map Editor, QuArK)
Cub, 505
Portal, 510
Roadbed, 506
Subtraction, 509
buddycount parameter, 229
buddylist parameter, 229
BuildTaggedString() function, 676
bulletin boards, as support infrastructure, 22
bump mapping, 101–102

buttons
Animate (Keyframer tool), 404
GuiButtonCtrl class, 340–341
MilkShape toolbox
Groups tab, 401
Joints tab, 403
Materials tab, 402
Model tab, 399–400
Mirror (Mission Area Editor), 594
Play (Audacity tool), 541
Record (Audacity tool), 540–541
buttonType property, 164
byte code, 134

C

CalcExplosionCoverage() function, 676
Call() function, 676
calling functions, 70, 126
Camera class, 728
Camera menu (Mission Editor), 589
Camera To Selection command (World menu), 590
CameraDecay command, 618
CameraLag command, 618
CameraMaxDist command, 154, 184, 618
CameraOffset command, 618
CameraRoll command, 618
campfire effects, 595–598
Can Line Stipple option (Preferences dialog box), 405
Cancel() function, 676
CancelServerQuery() function, 677
Canvas module, client-side, 259–261
Canyon Fractal function, 537
Capture the Flag, 3
Cartesian coordinates, 95
carving, bump mapping, 101
case keyword, 54, 735
case-sensitivity, variables, 54
celsalute animation sequence, torque-supported, 445
celwave animation sequence, torque-supported, 445
center property, 164
Center World, 6
CFXweb Web site, 743
Chain Reaction, 6
character models
animation
blended, 445
joint rotation, bone movement during, 445
rigging, 444
skeletal animation, 446–458
torque-supported sequences, 444–445
arbitrary extrusion, 417
arms, 433–437
arms to torso, integrating, 438
axial extrusion, 416
continuous-mesh model, 417
head, 418–423
head to torso, matching, 429–430
hybrids, 417
incremental polygon construction, 415–416
legs, 430–432
legs to torso, integrating, 432
segmented-mesh model, 417
shape primitives, 415
skins, 438–443
topographical shape mapping, 417
torso models, 423–429
ChatBox interface, 636–640, 650–652
ChatMessage function, 267
ChatMessageAll function, 653
chatPageDown function, 640
cheating, online, fighting against, 209
check boxes, GUI, 335, 341–342
checkDismountPoint method, 616
checkpoints and laps, scoring, 622–625
CheckProgress method, 624
childMargin property, 343
chunked bitmaps, 337–338
classes
AIPlayer, 728
BanList, 728
Blender, 26
Camera, 728
Control, 25
Debris, 728
defined, 26
EditTSCtrl, 728
FileObject, 728
FlyingVehicle, 729
GameBase, 729
GameConnection, 729

GUI control classes, 337–345
GuiBitmapCtrl, 639, 729
GuiButtonCtrl, 632
GuiCanvas, 729
GuiControl, 630, 730
GuiEditCtrl, 730
GuiFilterCtrl, 730
GuiFrameSetCtrl, 730
GuiInspector, 730
GuiMessageVectorCtrl, 730
GuiNoMouseCtrl, 639
GuiPopUpMenuCtrl, 730
GuiScrollCtrl, 224
GuiSliderCtrl, 731
GuiTerrPreviewCtrl, 731
GuiTestListCtrl, 731
GuiTextEditCtrl, 224
GuiTreeViewCtrl, 732
HTTPObject, 732
InteriorInstance, 732
Item, 732
Lightning, 732
MessageVector, 732
PhysicalZone, 732
Player, 732
Precipitation, 732
Profile, 25
SceneObject, 732
ServerScreen, 642
ShapeBase, 732
ShapeBaseData, 734
SimpleNetObject, 734
Sky, 734
StaticShape, 735
superclass, 129
TCPObject, 735
Terraformer, 735
Trigger, 735
TriggerData, 735
WaterBlock, 735
WorldEditor, 735

className property, 154, 184
Clear button (MilkShape Joints tab), 403
Clear Empty function, 537
ClearTextureHolds() function, 677
clicking, mouse operations, 591
client control modules
control/client/client.cs, 160–164
control/client/interfaces/menuscreen.gui, 162
control/client/interfaces/playerinterface.gui, 165–168
control/client/interfaces/splashscreen.gui, 169
control/client/misc/presetkeys.cs, 171–174
control/client/misc/screens.cs, 169–171

client interfaces
ChatBox, 636–640, 650–652
FindServer, 635–636, 648–650
Host, 635, 647–648
MenuScreen, 632–634
MessageBox, 640–642, 652–655
SoloPlay, 634–635, 643

client modules
Canvas module, 259–261
discussed, 258–259
functions, list of, 269–270
messages module, 266–268
mission module, 261–262

client-only sounds, 560
client *versus* server design issues, 132–133
ClientCmdChatMessage function, 267
ClientCmdMissionStart function, 262
ClientConnection module, 250–256
clipColumnText property, 344
Clone Brush tool (Paint Shop Pro), 297–298
Close command (Audacity File menu), 546
Close method, 655
clothing, player skin example, 329–332
cloud layers, 521–523
code
byte code, 134
common, 129
control, 129, 139–140
executable, 51

code module, finding servers, 225–229
CodeGuru Web site, 743
coins, scoring, 625–627
CollapseEscape() function, 677
collisions
collision mesh, vehicle models, 476
Collision Mesh option (Torque Game Engine (DTS) Exporter dialog box), 459–460
CollisionTol command, 618
vehicle models, 612–613

color
Color command (UVMapper Edit menu), 408
Color dialog box (Paint Shop Pro), 280
color matching, photography, 354–355
colors property, 601
Colors tab (Materials palette), 291
fadeColor property, 530
columns property, 344
command interface control, 340
command property, 164, 340, 634
commands
Animation menu (MilkShape), 394
Audacity tool shortcut keys, 550
Camera menu (Mission Editor), 589
Edit menu
Audacity tool, 547
Configuration, 500
Duplicate, 432
Duplicate Selection, 390
Hide Selection, 389, 425
MilkShape 3D, 392
Mission Editor, 588
New UV Map, 387
UVMapper, 408
Effect menu (Audacity tool), 549
Face menu (MilkShape), 394
File menu
Audacity tool, 546
Import, 387
Merge, 429
MilkShape 3D, 392
Mission Editor, 588
Preferences, 404
Save As File, 507
Save Model, 387
UVMapper, 407
grep
overview, 32
in UltraEdit-32, 39–41
Help menu (UVMapper), 408
Project menu (Audacity tool), 548
Vertex menu (MilkShape 3D), 392
View menu (Audacity tool), 547
WheeledVehicleData, 618–619
Window menu (MilkShape), 397
World menu, 590
CommandToClient() function, 207–209, 250, 677
CommandToServer() function, 206–207, 242, 250, 678
comments, defined, 50
common code, 129
compilation errors, 82–83
Compile() function, 678
Compile Quake 1 MDL command (MilkShape Tools menu), 395
compound statements, 52
compression, lossy, 285
computation, 58
concatenation, strings, 59
conditional expressions, 71–73
Configuration command (Edit menu), 500
configuring
QuArK, 500–501
UltraEdit-32, 33–35
Connect() function, 212
console library, Torque Game Engine, 24
constantAcceleration property, 601
constantThumbHeight property, 225, 343
constraints, game design, 585–586
constructs, entities, 509–510
ContactTol command, 618
ContainerBoxEmpty() function, 678
ContainerFindFirst() function, 679
ContainerFindNext() function, 679
ContainerRayCast() function, 679
containers, parent controls, 346
ContainerSearchCurrDist() function, 679
ContainerSearchCurrRadiusDist() function, 680
ContainerSearchNext() function, 680
Content Editor, Torque GUI Editor, 345–346
continue keyword, 54, 735
continuous-mesh model, character models, 417
Control class, 25
control/client/client.cs module, 160–164
control/client/interfaces/menuscreen.gui module, 162
control/client/interfaces/playerinterface.gui module, 165–168
control/client/interfaces/splashscreen.gui module, 169
control/client/misc/presetkeys.cs module, 171–174
control/client/misc/screens.cs module, 169–171
control/client.cs module, 144–148
control code, 129, 139–140

control flow, Torque Game Engine, 23
Control Inspector, Torque GUI Editor, 346
control/main.cs module, 159
control modules. *See* modules
Control Panel command (MilkShape Window menu), 397
control/player.cs module, 151–153
control/server/misc/item.cs module, 197–202
control/server/players/player.cs module, 180–186
control/server/server.cs module, 175–180
control/server/weapons/crossbow.cs module, 190–197
control/server/weapons/weapon.cs module, 186–190
control/server.cs module, 149–151
Control Tree, Torque GUI Editor, 346
controls, GUI
discussed, 336–337
GuiButtonCtrl class, 340–341
GuiCheckBoxCtrl class, 341–342
GUIChunkedBitmapCtrl, 337–338
GUIControl class, 339
GuiScrollCtrl class, 342–343
GuiTextEditCtrl class, 344–345
GuiTextListCtrl class, 343–344
Convert Old MS3D Format command (MilkShape Tools menu), 395
coordinate systems
axis, 90
Cartesian coordinates, 95
Computer Graphics Aerobics, 90
left-handed, 90
object space, 90
rectangular coordinates, 95
right-handed, 90
vertices, 92
world space, 91
XYZ-axis system, 90–93
Copy command
Audacity Edit menu, 547
Mission Editor Edit menu, 588
Copy Keyframes command (MilkShape Animate menu), 394
covers, terrains, 369, 378–380
Cox, Steve (UVMapper program creation), 406
CRC (Cyclic Redundancy Check), 246
CreateCanvas() function, 680
createPlayer function, 610
createServer() function, 143, 240
cropping images, 355–357
cross-platform software, disadvantages, 11
CS Hand Offset option (Preferences dialog box), 405
Cube brush (Torque Map Editor, QuArK), 505
current frame number box (Keyframer tool), 404
Cut command
Audacity Edit menu, 547
Mission Editor Edit menu, 588
cuts, axial extrusion, 416
cycleGame function, 656
Cyclic Redundancy Check (CRC), 246
Cylinder button (MilkShape Model tab), 399
cylinder shapes
character models, head modes, 418–419
MilkShape 3D, 384–386
Cylinder tool (MilkShape), 485
Cylindrical Cap unwrapping method, 410, 412
Cylindrical unwrapping method, 409, 412

D

data blocks
defined, 108
overview, 128–129
sound, creating and programming, 550–555
data-manipulation tasks, 4
databases, support infrastructures, 22
datablock keyword, 735
datablock property, 530, 599
DbgSetParameters() function, 680
DeactivateDirectInput() function, 680
DeactivateKeyboard() function, 681
DeactivatePackage() function, 681
Death Match, 3
deaths
death animation, 444, 455–457
tracking, scoring techniques, 628–629
Debian Linux distribution, 11
Debris class, 728
Debug() function, 681
Debug_debugbreak() function, 681
debugging
best practices, 86–87
compilation errors, 82–83
problem solving techniques, 83–86
using trace() function, 141
Debug_testx86unixmutex() function, 681
Debug_testxunixmutex(), 681

decimal systems, 46
declarations, forward, 48
Decompile Genesis command (MilkShape Tools menu), 395
DecreaseFSAA() function, 682
DecreaseNPatch() function, 682
decrement operators, 62
dedicated servers, 230–232, 662
default keyword, 54, 735
defaultLineHeight property, 343
DefaultMessageCallback function, 268
Delete All command (MilkShape Edit menu), 392
Delete button (MilkShape)
Groups tab, 401
Materials tab, 402
Delete command (Audacity File menu), 547
Delete Keyframe command (MilkShape Animate menu), 394
Delete Selection command (MilkShape)
Edit menu, 392
World menu, 590
DeleteDataBlocks() function, 682
DeleteVariables() function, 682
Delta Force, 22, 217
DEM (Digital Elevation Model), 367
Density control (Paint Brush tool), 295
density property, 154, 184
DepthGradient property, 535
description property, 552
DestroyedLevel command, 618
DestroyServer function, 179, 240
Detag() function, 682
developers, roles of, 11–12
development, gaming industry, 1–2
dialog boxes
Add Noise (Paint Shop Pro), 286
Box Mapping, 409
Color (Paint Shop Pro), 280
Cylindrical, 409
Cylindrical Cap Mapping, 410
Find, 36–37
New Image (Paint Shop Pro), 279
Planar Mapping, 409
Preferences (MilkShape 3D), 404–406
Project Setup, 35
Replace, 37
Soft Focus, 314
Texture Coordinate Editor, 494–495
Torque Game Engine (DTS) Exporter
Animation Settings option, 460
Collision Mesh option, 459–460
Other Settings option, 460
Diffuse button (MilkShape Materials tab), 402
Digital Elevation Model (DEM), 367
digital images *versus* film, 352–353
Dimensions command (UVMapper Help menu), 408
direct messaging
CommandToClient function, 207–209
CommandToServer function, 206–207
discussed, 205
online cheating, fighting against, 209
direct movement, translation, 105–106
Direct3D, advantages/disadvantages, 10
directed graphs, 103
DisabledLevel command, 618
DisableMouse() function, 683
dismounting, 615–616
display options, layers, 292
distant object textures, 278
DistortGridScale property, 535
distortion, 515, 518
DistortMag/DistortTime property, 535
distributions, Linux operating systems, 11
Divide Edge command (MilkShape Vertex menu), 393
DnetSetLogging() function, 683
do keyword, 54, 735
Doctor Dobb's Journal Web site, 744
DoExitGame() function, 657
DoJump function, 175
dollar sign ($), 54
DoPitch function, 175
DoScore method, 624
double-quoted strings, 124
double-sided surface, 94
double-slash operator (//), 50
down-stream industry, 2
DoYaw function, 175
Drag command, 618
drag property, 154, 184
dragCoefficient property, 601
Drop Camera At Player command (Mission Editor Camera menu), 589
Drop Player At Camera command (Mission Editor Camera menu), 589

Drop Selection command (World menu), 590
DumpConsoleClasses() function, 683
DumpMemSnapshot() function, 683
DumpNetStringTable() function, 683
DumpResourceStats() function, 684
DumpTextureStats() function, 684
Dungeons & Dragons, 4
Duplicate command
  Edit menu, 432
  File menu, 547
Duplicate Selection command (Edit menu), 390, 392
dynamic objects, shapes as, 104

E

Echo command (Audacity Effect menu), 549
Echo() function, 83, 684
EchoInputState() function, 684
edges, shapes, 94
Edit Area check box (Mission Area Editor), 594
edit boxes, GUI, 335
Edit-Compile-Link-Run cycle, 48
Edit ID3 Tags command (Audacity Project menu), 548
Edit menu
  Audacity tool, 545, 547
  Configuration command, 500
  Duplicate command, 432
  Duplicate Selection command, 390
  Hide Selection command, 389, 425
  MilkShape 3D, 391–392
  Mission Editor, 588
  New UV Map command, 387
  UVMapper, 407–408
editors
  Mission Area Editor, 593–594
  Mission Editor, 587–589
  programming features, 31
  Terrain Editor, 534–537, 590–592
  Terrain Terraform, 592
  Terrain Texture Editor, 592–593
  text, 31
  UltraEdit-32
    bookmark capabilities, 41
    configuring, 33–35
    discussed, 31
    Find in Files feature, 38
    grep command capabilities, 39–41
    Help feature, 45
    installing, 32
    macro commands, 43
    program setup, 32
    Project Setup dialog box, 35
    projects and files, setting up, 32
    search capabilities, 35–37
    UEPM (UltraEdit Project Maker), 32
  World Editor, 589–590
EditTSCtrl class, 728
Effect menu (Audacity tool), 545, 549
ejectionOffset property, 600
ejectionPeriodMS property, 600
ejectionVelocity property, 600
elements and arrays, 56
else keyword, 54, 735
Emaga4
  navigation keys, 154
  properties, 154
Emaga5
  key bindings, 202
  properties, 184
emap property, 154, 184
Emissive button (MilkShape Materials tab), 402
emitter property, 599
emitters, particles, 604
EnableMouse() function, 684
EnableWinConsole() function, 685
endGame function, 656
EndMission function, 246, 258
engine vehicle sound, 576
EngineBrake command, 618
EngineSound command, 619
EngineTorque command, 618
ENormals flag (special materials, MilkShape), 462
entities, 509–510
enumerate property, 344
Envelope tool (Audacity toolbar), 543
environment mapping, 102, 185, 461–462
environmental sound effects, 578–579
envMapIntensity property, 535
envMapOverTexture property, 535
envMapUnderTexture property, 535
Eraser tool (Paint Shop Pro), 298
Error() function, 685
errors
  compilation, 82–83
  error function, 83

escapeCommand property, 642, 655
Eval method, 655
evaluation
expressions, 52
for loop, 65
order of, 62
precedence, 56
events
list of, 24
simulation of objects, 24
triggers
creating, 620–621, 623
kill tracking, 629–630
onEnterTrigger, 621
onLeaveTrigger, 621
onTickTrigger, 621
scoring, 622–629
SetImageTrigger method, 629
Everquest, 22
examples. *See* programs; scripts
Excavate function, 537
exclamation point (!), 73
Exec() function, 566, 685
executable code, 51
Exit command
Audacity File menu, 546
MilkShape File menu, 392
ExpandEscape() function, 685
ExpandFilename() function, 552, 685
exploded skyboxes, 514
Export as WAV command (Audacity File menu), 546
Export command (MilkShape File menu), 392
Export() function, 686
Export Labels command (Audacity File menu), 546
export options (MilkShape special materials), 461
Export Selection as WAV command (Audacity File menu), 546
Export Terraform Bitmap command (Mission Editor File menu), 588
Export UVs command (UVMapper File menu), 407
expressions
branching, 74–75
conditional, 71–73
defined, 52
evaluations, 52
logical, 73–74
operands, 72
order of evaluation, 62
regular, 39
statements, 52
variables, 53
extent property, 164, 338, 652
external methods, terrains, 367
extrapolation, 28
Extrude button (MilkShape Model tab), 399

F

fabric textures, 362
Face button (MilkShape Model tab), 399
face handlers, 505
Face menu (MilkShape 3D), 393–394
Face To Front command (MilkShape Face menu), 394
Fade In command (Audacity File menu), 549
Fade Out command (Audacity File menu), 549
fadeColor property, 530
fake phone shading, 100
Falcon 4, 6
fall animation sequence, torque-supported, 444
false keyword, 54, 735
fast phong shading, 100
fBm Fractal function, 537
fenders, vehicle models, 473–474
FFT Filter command (Audacity File menu), 549
fidelity
simulator games, 6
terrains, 366
field of view (FOV), 515
File menu
Audacity tool, 545–546
Import command, 387
Merge command, 429
MilkShape 3D, 391–392
Mission Editor, 588
Preferences command, 404
Save As File command, 507
Save Model command, 387
UVMapper tool, 407
File Tree View (UltraEdit-32), 33
FileBase() function, 686
FileExt() function, 686
FileName() function, 686
FileObject class, 728
FilePath() function, 686

files
listed files, Find in Files feature, 38
open files, Find in Files feature, 38
project files, Find in Files feature, 38
setting up, UltraEdit-32, 32
texture files, saving, 284–285
Fill tool (Paint Shop Pro), 516–517
film *versus* digital images, 352–353
Filter function, 537
Filter Textures option (Preferences dialog box), 405
filterflags parameter, 229
Find dialog box, 36–37
Find in Files feature, UltraEdit-32, 38
FindFirstFile() function, 647, 687
FindNextFile() function, 646, 648, 687
FindServer interface, 635–636, 648–650
First-Person Point-of-View (1st PPOV) games, 3, 662–663
First-Person Shooter (FPS) games, 3
FirstWord() function, 687
Fit in Window command (Audacity View menu), 547
fitParentWidth property, 344
flags
flags parameter, 228
mesh option (MilkShape special materials), 462–463
option (MilkShape special materials), 461–462
flat shading, 99, 469
Flatten function, 537
flesh-tone RBG settings, 324
flipCode Web site, 744
flipping objects, 393
Float command (Audacity View menu), 547
FlowAngle/FlowRate property, 535
FlushTextureCache() function, 687
FlyingVehicle class, 729
fog effects, 523
folders, game root, 129–130
fonts, text, 306
footstep sounds, 560–563
for keyword, 54, 735
for loop, 65–66
forums, as support infrastructure, 22
forward declarations, 48
forward slash (/), 61
FOV (field of view), 515
FPS (First-Person Shooter) games, 3, 662–663
frames, 339, 382
freedom of terrains, 366
Freehand Selection tool (Paint Shop Pro), 300
FreeMemoryDump() function, 687
freestanding particles, 595
Freeverb2 command (Audacity File menu), 549
freeware and shareware tools, 750–752
front view, MilkShape 3D, 382
full transformation, 97
function block, 51
function body, 51
function headers, 51
function keyword, 54, 735
functions. *See also* methods
activateDirectInput(), 667
activateKeyboard(), 667
activatePackage(), 667
Add Dirt, 537
AddCardProfile(), 668
addMaterialMapping(), 668
addMessageCallback, 268
AddOSCardProfile(), 669
AddTaggedString(), 669
AddToServerGuidList, 241
Adjust Height, 537
Adjust Selection, 537
alGetString(), 669
alListener3f(), 670
AllowConnections, 670
alxCreateSource(), 670
alxGetChannelVolume(), 671
alxGetListener3f(), 671
alxGetListenerf(), 671
alxGetListeneri(), 671
alxGetSource3f(), 672
alxGetSourcef(), 672
alxGetSourcei(), 672
alxListener(), 673
alxListenerf(), 552
alxIsPlaying(), 673
alxPlay(), 552, 673
alxSetChannelVolume(), 673
alxSource3f(), 674
alxSourcef(), 674
alxSourcei(), 675
alxStop(), 675
alxStopAll(), 675

arguments, 70
Backtrace(), 675
BackUp, 175
Bitmap, 537
Blend, 537
BuildTaggedString(), 676
CalcExplosionCoverage(), 676
Call(), 676
calling, 70, 126
Cancel(), 676
CancelServerQuery(), 677
Canyon Fractal, 537
ChatMessage, 267
ChatMessageAll, 653
chatPageDown, 640
Clear Empty, 537
ClearTextureHolds(), 677
client module functions, list of, 269–270
ClientCmdChatMessage, 267
ClientCmdMissionStart, 262
CollapseEscape(), 677
CommandToClient, 207–209, 250
CommandToClient(), 677
CommandToServer(), 206–207, 242, 250, 678
Compile(), 678
Connect(), 212
ContainerBoxEmpty(), 678
ContainerFindFirst(), 679
ContainerFindNext(), 679
ContainerRayCast(), 679
ContainerSearchCurrDist(), 679
ContainerSearchCurrRadiusDist(), 680
ContainerSearchNext(), 680
CreateCanvas(), 680
createPlayer, 610
CreateServer(), 143, 240
cycleGame, 656
datablocks, 108
DbgSetParameters(), 680
DeactivateDirectInput(), 680
DeactivateKeyboard(), 681
DeactivatePackage(), 681
Debug(), 681
Debug_degub_debugbreak(), 681
DecreaseFSAA(), 682
DefaultMessageCallback, 268
defined, 26, 66
DeleteDataBlocks(), 682
DeleteVariables(), 682
DescreaseNPatch(), 682
DestroyServer, 179, 240
Detag(), 682
DisableMouse(), 683
DnetSetLogging(), 683
DoExitGame(), 657
DoJump, 175
DoPitch, 175
DoYaw, 175
DumpConsole(), 683
DumpMemSnapshot(), 683
DumpNetStringTable(), 683
DumpResourceStats(), 684
DumpTextureStats(), 684
Echo(), 83, 684
EchoInputState(), 684
EnableMouse(), 684
EnableWinConsole(), 685
endGame, 656
EndMission, 246, 258
error, 83
Error(), 685
Excavate, 537
Exec(), 685
ExpandEscape(), 685
ExpandFilename(), 552, 685
Export(), 686
fBm Fractal, 537
FileBase(), 686
FileExt(), 686
FileName(), 686
FilePath(), 686
Filter, 537
FindFirstFile(), 647, 687
FindNextfile(), 646, 648, 687
FirstWord(), 687
Flatten, 537
FlushTextureCache(), 687
FreeMemoryDump(), 687
Fruit program example, 67–69
GameConnection(), 143
GetBoxCenter(), 688
GetBuildString(), 688
GetCompileTimeString(), 688
GetControlObject Altitude(), 688
GetControlObjectSpeed(), 688
GetDesktopResolution(), 689

GetDisplayDeviceList(), 689
GetField(), 689
GetFieldCount(), 689
GetFields(), 690
GetFileCount(), 690
GetFileCRC(), 690
getGroup, 652
getHelp, 634
GetJoystickAxes(), 690
GetMaxFrameAllocation(), 690
getMissionDisplayName, 646–647
GetModPaths(), 691
GetRandom(), 691
GetRandomSeed(), 691
GetRealTime(), 691
GetRecord(), 691
GetRecordCount(), 692
GetRecords(), 692
GetResolution(), 692
GetResolutionList(), 692
GetServerCount(), 692
GetSimTime(), 693
GetSubStr(), 693
GetTag(), 693
GetTaggedString(), 693
GetTerrainHeight(), 693
GetVersionNumber(), 694
GetVersionString(), 694
GetVideoDriverInfo(), 694
GetWord(), 652, 694
GetWordCount(), 694
GetWords(), 695
GLEnableLogging(), 695
GLEnableMetrics(), 695
GLEnableOutline(), 695
GoAhead, 175, 562–563
GoLeft, 175
GoRight, 175
GoToWebPage(), 695
Hydraulic Erosion, 537
IncreaseFSAA(), 696
IncreaseNPatch(), 696
InitBaseClient(), 143, 239
InitBaseServer(), 143, 239
InitCanvas(), 143
InitContainerRadiusSearch(), 696
initialControlSet(), 215
InitializeClient(), 143, 217–218, 642
InitializeServer(), 143
InputLog(), 696
IsDemoRecording(), 696
IsDeviceFullScreenOnly(), 697
IsEventPending(), 697
IsFile(), 697
IsFullScreen(), 697
IsJoystickDetected(), 697
IsKoreanBuild(), 698
IsNameUnique, 255
IsObject(), 698
IsPackage(), 698
IsPointInside(), 698
IsWriteableFileName(), 698
LaunchDedicatedServer(), 699
LaunchGame(), 161
LightScene(), 265, 699
LoadMission, 246, 250, 258
LoadMissionStage2, 246, 250
lockMouse(), 699
ltrim(), 699
mAbs(), 699
mAcos(), 700
main(), 51
makeTestTerrain(), 700
mAsin(), 700
mAtan(), 700
MathInit(), 701
MatrixCreate(), 701
MatrixCreateFromEuler(), 701
MatrixMulPoint(), 701
MatrixMultiply(), 702
MatrixMulVector(), 702
mCeil(), 702
mCos(), 702
mDegToRad(), 702
member, 127
mFloadLength(), 703
mFloor(), 703
MissionStartPhase, 250
mLog(), 703
MoveShape(), 109, 117
mPow(), 703
mRadToDeg(), 703
msg(), 704
mSin(), 704
mSolveCubic(), 704
mSolveQuadratic(), 704

mSolveQuartic(), 704
mSqrt(), 705
mTan(), 705
nameTolD(), 705
nCycleExec, 656
nextResolution(), 705
nextToken(), 706
objects, 126
onChatMessage, 268
onConnect(), 213
onConnectionAccepted(), 213
onConnectionDropped(), 214
onConnectionError(), 214
onConnectionRequest(), 212
onConnectionTimedOut(), 214
onConnectRequestRejected(), 213–214
onConnectRequestTimedOut(), 213
onCyclePauseEnd(), 656
onDataBlockObjectReceived(), 216, 265
onDataBlocksDone(), 216
onDrop(), 215
OnExit(), 140
onFileChunkReceived(), 216
onGhostAlwaysObjectReceived(), 216
onGhostAlwaysStarted(), 217
OnMissionDownloadPhase, 265
OnMissionEnded, 258
OnMissionLoaded, 179, 258
OnServerCreated(), 178, 258, 595, 620
OnServerDestroyed, 258
OnStart(), 140, 159, 231
OpenAlInitDriver, 706
OpenALShutdownDriver(), 706
Paint Material, 537
PanoramaScreenShot(), 706
parameters, 71
ParseArgs, 138, 230
pathOnMissionLoadDone(), 707
PermDisableMouse(), 707
PlayDemo(), 707
PlayJournal(), 707
PortInit, 240
PrevResolution(), 707
problem decomposition, 66
ProfilerDump(), 708
ProfilerDumpToFile(), 708
ProfilerEnable(), 708
ProfilerMarkerEnable(), 708
PurgeResources(), 708
QueryMasterServer(), 709
QueryStatus(), 225
Quit(), 709
RedbookClose(), 710
RedbookGetDeviceCount(), 710
RedbookGetDeviceName(), 710
RedbookGetLastError(), 710
RedbookGetTrackCount(), 710
RedbookGetVolume(), 711
RedbookOpen(), 711
RedbookPlay(), 711
RedbookSetVolume(), 711
RedbookStop(), 711
ReferenceDistance, 564
RemoveField(), 712
RemoveRecord(), 712
RemoveTaggedString(), 712
RemoveWord(), 712
ResetLighting(), 712
ResetMission, 246, 258
ResetServer, 241
RestWords(), 713
Rigid Multifractal, 537
Rtrim(), 713
SaveJournal(), 713
SceneLightingComplete, 265
Schedule(), 115, 178–179, 713
ScreenShot(), 713
Select, 537
SendMacro(), 206
server modules, list of, 270–272
ServerMessage, 268
serverPlay3D, 560, 562
Set Empty, 537
Set Height, 537
SetDefaultFov(), 714
SetDisplayDevice(), 714
SetEchoFileLoads(), 714
SetField(), 714
SetFov(), 715
SetFSAA(), 715
SetInteriorFocusedDebug(), 715
SetInteriorRenderMode(), 715
setLagIcon(), 215
SetLogMode(), 715
SetModPaths(), 716
SetNetPort(), 716

SetNPatch(), 716
SetOpenGLAnisotropy(), 716
SetOpenGLInteriorMipReduction(), 716
SetOpenGLMipReduction(), 717
SetOpenGLSkyMipReduction, 717
SetOpenGLTextureCompressionHint(), 717
SetRandomSeed(), 717
SetRecord(), 717
SetResolution(), 718
SetScreenMode(), 718
SetServerInfo, 718
SetShadowDetailLevel(), 718
SetVerticalSync(), 718
SetWord(), 719
SetZoomSpeed(), 719
ShapeBaseImageData, 570
ShowMenuScreen(), 161
Sinus, 537
Smooth, 537
Smooth Ridges, 537
Smooth Water, 537
Smoothing, 537
SpamAlert, 242
StartGame, 178
StartHeartbeat(), 719
StartRecording(), 719
StopHeartbeat(), 719
StopRecording(), 720
StopServerQuery(), 720
Strchr, 720
Strcmp(), 720–721
StripChars(), 721
StripMLControlChars(), 721
StripTrailingSpaces(), 721
Strlen(), 721
Strlwr(), 722
Strpos(), 722
Strreplace, 722
Strstr(), 722
StrToPlayerName(), 723
Strupr(), 723
SwitchBitDepth(), 723
TellAll(), 207
TelnetSetParameters(), 723
Terrain File, 537
TestShape(), 109
that return values, 71
Thermal Erosion, 537
Toggle3rdPPOVLook, 175
ToggleFullScreen(), 723
ToggleInputState(), 724
ToggleMessageBox, 637
ToggleNPatch(), 724
Trace(), 141, 724
Trim(), 724
Turbulence, 537
UpdateLightingProgress, 266
Usage(), 138
utility functions, Torque Game Engine, 25
ValidateMemory(), 724
VectorAdd(), 725
VectorCross(), 725
VectorDist(), 725
VectorDot(), 725
VectorLen(), 725
VectorNormalize(), 726
VectorOrthoBasis(), 726
VectorScale(), 726
VectorSub(), 726
VideoSetGammaCorrection(), 260, 726
Warn(), 83, 727
WeaponImage, 570
without parameters, 70
without return values, 70–71
funForce property, 154

G

Gamasutra Web site, 744
game design
constraints, 585–586
requirements specification, 584–585
game engines
block diagram, 16
overview, 16–17
Game module, server-side, 256–258
game root folders, 129–130
GameBase class, 729
GameConnection class, 729
GameConnection() function, 143
GameConnection object, 211–212
initialControlSet() function, 215
onConnect() function, 213
onConnectionAccepted() function, 213
onConnectionDropped() function, 214
onConnectionError() function, 214
onConnectionRequest() function, 212

onConnectionTimedOut() function, 214
onConnectRequestRejected() function, 213–214
onConnectRequestTimedOut() function, 213
onDataBlockObjectReceived() function, 216
onDataBlocksDone() function, 216
onDrop() function, 215
onFileChunkReceived() function, 216
onGhostAlwaysObjectReceived() function, 216
onGhostAlwaysStarted() function, 217
setLagIcon() function, 215
GameDev.net Web site, 744
Gamer's Technical Resources Web site, 745
games
1st PPOV (First-Person Point-of-View), 3
3rd PPOV (Third-Person Point-of-View), 3
Attach & Defend, 3
Blackhawk Down, 22
Capture the Flag, 3
Center World, 6
Chain Reaction, 6
Death Match, 3
Delta Force, 22, 217
Dungeons & Dragons, 4
Everquest, 22
Falcon 4, 6
FPS (First-Person Shooter), 3, 662–663
genres
action games, 3
adventure games, 3–4
retail games, 2
RTS (Real-Time Strategy), 7
simulator games, 6
sports games, 7
strategy games, 7–8
Half-Life 2, 17
The Incredible Machine series, 6
King-of-the-Hill, 3
MarbleBlast, 5
Maximum Football, 7
mazes, 5–6
Myrmidon, 5
platforms, 8–11
puzzles, 5–6
Quake 3, 17
RPGs (role playing games), 4–5
SimCity series, 8
ThinkTanks, 3
Tribes 2, 17
Unreal II, 17
World War II Online, 22
gaps in map box mapping option, 411
gaps in map cylindrical cap mapping option, 412
gaps in map cylindrical mapping option, 412
gaps in map planar mapping option, 411
gaps in map spherical mapping option, 413
GarageGames Web site, 27, 742
genres
action games, 3
adventure games, 3–4
retail games, 2
RTS (Real-Time Strategy), 7
simulator games
overview, 6
strategic simulations, 8
sports games, 7
strategy games, 7–8
geometric center of objects, 90
GeoSphere button (MilkShape Model tab), 399
GetBoxCenter() function, 688
GetBuildString() function, 688
GetCompileTimeString() function, 688
GetControlObjectAltitude() function, 688
GetControlObjectSpeed() function, 688
GetDesktopResolution() function, 689
GetDisplayDeviceList() function, 689
GetField() function, 689
GetFieldCount() function, 689
GetFields() function, 690
GetFileCount() function, 690
GetFileCRC() function, 690
getGroup function, 652
getHelp function, 634
GetJoystickAxes() function, 690
GetMaxFrameAllocation() function, 690
getMissionDisplayName function, 646–647
GetModPaths() function, 691
GetRandom() function, 691
GetRandomSeed() function, 691
GetRealTime() function, 691
GetRecord() function, 691
GetRecordCount() function, 692
GetRecords() function, 692
GetResolution() function, 692
GetResolutionList() function, 692
GetServerCount() function, 692

GetSimTime() function, 693
GetSubStr() function, 693
GetTag() function, 693
GetTaggedString() function, 693
GetTerrainHeight() function, 693
getTransform() method, 109, 562
GetVersionNumber() function, 694
GetVersionString() function, 694
GetVideoDriverInfo() function, 694
GetWord() function, 652, 694
GetWordCount() function, 694
GetWords() function, 695
glass textures, 278
GLEnableLogging() function, 695
GLEnableMetrics() function, 695
GLEnableOutline() function, 695
global scope, 54
Gnometech Web site, 742
GNU General Public License, 754
GoAhead function, 175, 562–563
GoLeft function, 175
GoRight function, 175
GoToWebPage() function, 695
gouraud shading, 99–100
Graphical User Interface (GUI)
controls
discussed, 336–337
GuiButtonCtrl class, 340–341
GuiCheckBoxCtrl class, 341–342
GUIChunkedBitmapCtrl, 337–338
GUIControl class, 339
GuiScrollCtrl class, 342–343
GuiTextCtrl class, 339–340
GuiTextEditCtrl class, 344–345
GuiTextListCtrl class, 343–344
discussed, 335
overview, 19
Torque GUI Editor
Content Editor, 345–346
Control Inspector, 347
Control Tree, 346
creating interfaces using, 348–349
keyboard commands, 348
launching, 345
Tool Bar, 347
graphs
directed, 103
scene, 103–104
gravityCoefficient property, 601
grep command
overview, 32
in UltraEdit-32, 39–41
Grid Size option (Preferences dialog box), 405
group nodes, scene graphs, 103
Group Selector Box button (MilkShape Groups tab), 401
Groups tab (MilkShape toolbox), 400–401
gtype parameter, 228
GUI (Graphical User Interface)
controls
discussed, 336–337
GuiButtonCtrl class, 340–341
GuiCheckBoxCtrl class, 341–342
GUIChunkedBitmapCtrl, 337–338
GUIControl class, 339
GuiScrollCtrl class, 342–343
GuiTextCtrl class, 339–340
GuiTextEditCtrl class, 344–345
GuiTextListCtrl class, 343–344
discussed, 335
overview, 19
Torque GUI Editor
Content Editor, 345–346
Control Inspector, 347
Control Tree, 346
creating interfaces using, 348–349
keyboard commands, 348
launching, 345
Tool Bar, 347
GuiBitmapCtrl class, 639, 729
GuiButtonCtrl class, 340–341, 632
GuiCanvas class, 729
GuiCheckBoxCtrl class, 341–342
GUIChunkedBitmapCtrl class, 337–338
GuiContentProfile property, 163
GuiControl class, 639, 730
GuiDefaultProfile method, 578–579
GuiEditCtrl class, 730
GuiFilterCtrl class, 730
GuiFrameSetCtrl class, 730
GuiInspector class, 730
GuiMessageVectorCtrl class, 730
GuiNoMouseCtrl class, 639
GuiPopUpMenuCtrl class, 730
GuiScrollCtrl class, 224, 342–343
GuiSliderCtrl class, 731

GuiTerrPreviewCtrl class, 731
GuiTextCtrl class, 339–340
GuiTextEditCtrl class, 224, 344–345
GuiTextListCtrl class, 731
GuiTreeViewCtrl class, 732
gun creation
model building, 490–494
skins, 494–495
sound effects, 565–572
testing, 495–496
gunshot sound-effect wave form, 21

H

hair texture, player skin example, 327–328
Half-Life 2, 17
Half-Life command (MilkShape Tools menu), 395
Hall of Worlds Web site, 742
handles
defined, 505
face, 505
objects, 125-126
hands, player skin example, 329
HardImpactSound command, 619
hardImpactSound property, 576
HardImpactSpeed command, 618
Hardness control (Paint Brush tool), 295
head animation
character modeling, 418–423
head and neck, player skin example, 322–326
head animation sequence, torque-supported, 444
head to torso model, character modeling, 429–430
skeletal animation, 447–448, 454–455
header blocks, 50
Heads Up Display (HUD), 19, 168
headside animation sequence, torque-supported, 444
Health Kit model, 479–481
height-maps, terrains, 367–368
height property, 164
Hello World program example, 49–51
help
Help feature, UltraEdit-32, 45
Help menu (UVMapper), 407, 409
Usage() function, 138
Hense, Michael (*Center World*), 6
hero rigging, skeletal animation, 451
hexadecimal systems, 46
hidden lines, 94
hidden surfaces, 94
Hide button (MilkShape Groups tab), 401
Hide Selection command
Edit menu, 389, 392, 425
World menu, 590
high-constrasting shading, 351
high-level languages, 47
highlight maps, 100
highlighting, syntax highlighting, 31
holes, bump mapping, 101
home game consoles, 8
horizSizing property, 164
Host interface, 635, 647–648
hosted servers, 661–662
hot keys, UVMapper tool, 410
Hot keys command (UVMapper Help menu), 408
house structures, 508–511
hScrollBar property, 225, 343
HTTPObject class, 732
HUD (Heads Up Display), 19, 168
hybrids, character models, 417
Hydraulic Erosion function, 537

I

identifier token, 737
identifiers, variables, 54
iDevGames Web site, 745, 749
idle animation, skeletal animation, 451–452
idle engine sound, 576
if-else statements, 76–79
if keyword, 54, 735
if statement
nested if statements, 79–80
overview, 75–76
Illum flag (special materials, MilkShape), 462
image editing tools, 751
images
bitmap *versus* vector, 288–289
creating masks from, 302–303
cropping, 355–357
digital *versus* film, 352–353
rotating, 304–305
scaling, 303–304
sizes, changing, 305
tiling, 359–360

Import Audio command (Audacity Project menu), 548
Import command (File menu), 387, 392
Import Frame option (Preferences dialog box), 405
Import Labels command (Audacity Project menu), 548
Import MIDI command (Audacity Project menu), 548
Import Raw Data command (Audacity Project menu), 548
Import Terraform Data command (Mission Editor File menu), 588
Import Texture Data command (Mission Editor File menu), 588
Import UVs command (UVMapper File menu), 407
IncreaseFSAA() function, 696
IncreaseNPatch() function, 696
increment operators, 62
incremental polygon construction, 415–416
indenting, readability issues, 86
index of arrays, 56
infinite loops, 65
inheritedVelFactor property, 601
InitBaseClient() function, 143, 239
InitBaseServer() function, 143, 239
InitCanvas() function, 143
InitContainerRadiusSearch() function, 696
initialControlSet() function, 215
initialization
- control code, 139–140
- for loop, 65
- overview, 58
- program example, 141–143
- script for, 235–240

InitializeClient() function, 143, 217–218, 642
InitializeServer() function, 143
input model, Torque Game Engine, 24
InputLog() function, 696
insert credit Web site, 745
Insert Silence command (Audacity File menu), 547
installing
- Audacity tool, 540
- MilkShape 3D, 381–382
- Paint Shop Pro, 279
- QuArK, 500
- Torque, 29
- UltraEdit-32, 32

instance of objects, 125
instantiation, 125
Integration command, 618
interface module, finding servers, 218–223
interface sound, 578–579
interfaces
- ChatBox, 636–640, 650–652
- creating, using Torque GUI Editor, 348–349
- FindServer, 635–636, 648–650
- Host, 635, 647–648
- MenuScreen, 632–634
- MessageBox, 640–642, 652–655
- SoloPlay, 634–635, 643

interior library, Torque Game Engine, 27
interior structures, 499
InteriorInstance class, 732
InteriorInstance object, 125
internal methods, terrains, 368
International Game Developers Center Web site, 745
interpolation, 28
Invert command (Audacity File menu), 549
irregular textures, 360
IsDemoRecording() function, 696
IsDeviceFullScreenOnly() function, 697
IsEventPending() function, 697
IsFile() function, 697
IsFullScreen() function, 697
IsJoystickDetected() function, 697
IsKoreanBuild() function, 698
isLooping property, 551
IsNameUnique function, 255
IsObject() function, 698
IsPackage() function, 698
IsPointInside() function, 698
IsWriteableFileName() function, 698
Item class, 732
iteration, loops, 65
iterators and loops, 56

J

JetEnergyDrain command, 619
JetForce command, 619
JetSound command, 619
joint animation, bone movement during, 445
Joint button (MilkShape Model tab), 400
Joint Photographic Experts Group (JPEG), 285–288

Joint Selector Box button (MilkShape Joints tab), 403
Joint Size option (Preferences dialog box), 406
Joint tool (MilkShape), 617
Joints tab (MilkShape 3D), 403
JPEG (Joint Photographic Experts Group), 285–288
jump animation sequence, torque-supported, 444
jumpEnergyDrain property, 184
jumpForce property, 154, 184
jumpSurfaceAngle property, 154, 184

K

kerning, text, 306
key binding, 148, 202
key mapping, 148
keyboard and mouse operations, 591
keyboard commands, 348
keyframe slider (Keyframer tool), 404
Keyframer tool (MilkShape toolbox), 403–404
keyframes, 403
keywords
- break, 54, 735
- case, 54, 735
- continue, 54, 735
- datablock, 735
- default, 54, 735
- defined, 53
- do, 54, 735
- else, 54, 735
- false, 54, 735
- for, 54, 735
- function, 54, 735
- if, 54, 735
- new, 54, 735
- package, 736
- return, 54, 736
- switch, 54, 736
- true, 54, 736
- while, 54, 736

kill tracking, event triggers, 629–630
King-of-the-Hill, 3
koob utility, 555–558

L

Label tool (Audacity tool), 544
lag, 133
lambert shading, 99
land animation sequence, torque-supported, 444
laps and checkpoints, scoring, 622–625
latency problems, 28
LaunchDedicatedServer() function, 699
LaunchGame() function, 161
layers
- active, 293
- adjustment, 292
- background, 292
- creating, 292
- display options, 292
- mask, 291–292, 301–302
- merging, 306
- naming, 292
- raster, 291
- saving, 294
- shifting, 419
- vector, 291

leading text, 306
leaf nodes, scene graphs, 103
left-handed coordinate system, 90
left property, 164
legs, character modeling, 430–432
lifetimeMS property, 600–601
lifetimeVarianceMS property, 600–601
light maps, 100
lighting
- photography, 355
- SceneLightingCompleter function, 265
- UpdateLightingProgress function, 266

Lightning class, 732
lightning effects, 529–531
light_recoil animation sequence, torque-supported, 445
LightScene function, 265
LightScene() function, 265, 699
lines
- hidden lines, 94
- straight, 294

Linux Game Development Center Web site, 745
Linux Game Tome Web site, 745
Linux operating systems
- discussed, 10–11
- distributions, 11

Linux tool source Web sites, 749
listed files, Find in Files feature, 38
Load Model command (UVMapper File menu), 407

LoadMission function, 246, 250, 258
LoadMissionStage2 function, 246, 250
local scope, 54
Lock Selection command (World menu), 590
lockMouse() function, 699
logical expressions
examples of, 74
AND operator, 73
overview, 73–74
look animation, 444–445, 455
loops
for, 65–66
bounded, 64
defined, 64
infinite, 65
iteration, 65
iterators and, 56
while, 64–65
lossy compression, 285
lost data packets, 28
ltrim() function, 699

M

mAbs() function, 699
machine code, 45–46
Machinima.com Web site, 746
Macintosh
Macintosh Linux distribution, 11
Macintosh tool sources Web sites, 749
mAcos() function, 700
macro commands, UltraEdit-32, 43
main() function, 51
main screen, Audacity tool, 542–543
main.cs module, 139–140
makeTestTerrain() function, 700
Mandrake Linux distribution, 11
Manual Edit command (MilkShape Vertex menu), 393
map size box mapping option, 411
map size cylindrical mapping option, 412
map size planar mapping option, 411
map size spherical mapping option, 413
map2dif, QuArK reference, 767–768
maps, 100
MarbleBlast, 5
mAsin() function, 700
mask layers, 291–292
masks
creating, from images, 302–303
creating, from selections, 302
defined, 300
mask layers, creating, 301–302
Mass command, 618
mass property, 154, 184
MassBox command, 618
MassCenter command, 618
Master Gain tool (Audacity toolbar), 543
master servers, 217
mAtan() function, 700
material animation, 27
Material Preview button (MilkShape Materials tab), 402
Material Selector Box button (MilkShape Materials tab), 402
MaterialList property, 519
materials
special (MilkShape), 460–463
storm effects, 528–529
Materials palette (Paint Shop Pro), 290–291
Materials tab (MilkShape toolbox), 400, 402
MathInit() function, 701
MathWorld Web site, 746
MatrixCreate() function, 701
MatrixCreateFromEuler() function, 701
MatrixMulPoint() function, 701
MatrixMultiply() function, 702
MatrixMulVector() function, 702
maxBackwardSpeed property, 154, 184
maxbots parameter, 228
MaxDamage command, 618
maxDamage property, 154, 184
maxdrag property, 154, 184
MaxEnergy command, 619
maxEnergy property, 154, 184
maxForwardSpeed property, 154, 184
Maximum Football, 7
maxInv property, 184
maxJumpSpeed property, 154, 184
maxLength property, 340
maxplayers parameter, 228
maxSlideSpeed property, 154, 184
MaxSteeringAngle command, 618
maxVelocity property, 532
MaxWheelSpeed command, 619
mazes, 5–6

mCeil() function, 702
mCos() function, 702
mDegToRad() function, 702
Meade, Ian D. (UltraEdit-32), 31
member functions, 127
member variables, 127
menus
Audacity tool
Edit menu, 545, 547
Effect menu, 545, 549
File menu, 545–546
Project menu, 545, 548
View menu, 545, 547
GUI, 335
MilkShape 3D
Animate menu, 394
Edit menu, 391–392
Face menu, 393–394
File menu, 391–392
Tools menu, 395
Vertex menu, 391, 393
Mission Editor
Camera menu, 589
Edit menu, 588
File menu, 588
UVMapper tool
Edit menu, 407–408
File, 407
Help menu, 407, 409
Window (MilkShape), 397
World (World Editor), 590
MenuScreen interface, 632–634
Merge command (File menu), 392, 429
merging layers, 306
mesh animation, 27, 94
Mesh Factory Web site, 746
mesh option flags (MilkShape special materials), 462–463
message module
client-side, 266–268
server-side, 241–242
Message Panel (MilkShape 3D), 406
MessageBox interface, 640–642, 652–655
MessageDialog object, 174
MessageVector class, 732
messaging
direct
CommandToClient function, 207–209
CommandToServer function, 206–207
discussed, 205
online cheating, fighting against, 209
GameConnection messages, 211–212
GameConnection object, 211–212
initialControlSet() function, 215
onConnect() function, 213
onConnectionAccepted() function, 213
onConnectionDropped() function, 214
onConnectionError() function, 214
onConnectionRequest() function, 212
onConnectionTimedOut() function, 214
onConnectRequestRejected() function, 213–214
onConnectRequestTimedOut() function, 213
onDataBlockObjectReceived() function, 216
onDataBlocksDone() function, 216
onDrop() function, 215
onFileChunkReceived() function, 216
onGhostAlwaysObjectReceived() function, 216
onGhostAlwaysStarted() function, 217
setLagIcon() function, 215
metallic textures, 277, 362
methodologies, testing, 660
methods. *See also* functions
checkDismountPoint, 616
CheckProgress, 624
Close, 655
DoScore, 624
Eval, 655
exec(), 566
getTransform(), 562
GuiDefaultProfile, 578–579
mountObject, 614
mountPose, 611
of objects, 126
OnEscape, 655
onMount, 614
onServerQueryStatus, 650
OnWake, 642
Open, 654
Push, 171
Query, 650
schedule, 562
setAction, 615
SetImageTrigger, 629

setTransform(), 111
SpamMessageTimeout, 242
SpawnPlayer, 151
ToggleState, 655
Update, 650
UpdateLap, 621
mFloatLength() function, 703
mFloor() function, 703
MilkShape 3D
Box tool, 479–480
cylinder shapes, 384–386
Cylinder tool, 485
frames, 382
installing, 381–382
Joint tool, 617
menus
Animate, 394
Edit menu, 391–392
Face menu, 393–394
File menu, 391–392
Tools menu, 395
Vertex menu, 391, 393
Window menu, 397
Message Panel, 406
plug-ins, list of, 395–396
Preferences dialog box, 404–406
Scale tool, 493
special materials, 460–463
Sphere tool, 481
Texture Coordinate Editor, 406
toolbox
Groups tab, 400–401
Joints tab, 403
Keyframer tool, 403–404
Materials tab, 400, 402
Model tab, 398–400
Vertex tool, 488
views, 382–384
windows, 382
working environment, 382
zoom options, 383
MinAlpha/MaxAlpha property, 535
mincpu parameter, 228
minExtent property, 338
MinImpactSpeed command, 184, 618
MinJetEnergy command, 619
minJumpEnergry property, 184
minJumpSpeed property, 154, 184
minplayers parameter, 228
minRunEnergy property, 184
minVelocity property, 532
mipmapping, 102
MipZero flag (special materials, MilkShape), 462
Mirror button (Mission Area Editor), 594
mirroring
objects, 393
terrains, 594
Misc tab (Preferences dialog box), 404
Mission Area Editor, 593–594
Mission Editor, 587–589
mission module, client-side, 261–262
missiondownload module
client-side, 262–266
server-side, 246–250
MissionInfoObject control, 647
missionload module, server-side, 242–246
MissionStartPhase function, 250
mLog() function, 703
Model tab (MilkShape toolbox), 398–400
models
character animation
arbitrary extrusion, 417
arms, 433–437
arms to torso, integrating, 438
axial extrusion, 416
blended animation, 445
continuous-mesh model, 417
head, 418–423
head to torso, matching, 429–430
hybrids, 417
incremental polygon construction, 415–416
joint animation, bone movement during, 445
legs, 430–432
legs to torso, integrating, 432
rigging, 444
segmented-mesh model, 417
shape primitives, 415
skeletal animation, 446–458
skins, 438–443
topographical shape mapping, 417
torque-animation sequences, 444–445
torso models, 423–429
discussed, 19
Health Kit, 479–481
terrains, 20

vehicle models
body of vehicle, 467–472
collision mesh, 476
fenders, 473–474
mount nodes, 475–476
mounting, 611
sketches, 466–467
skins, 476
testing, 477–478
WheeledVehicleData property, 618–619
wheels, 476–477
module header blocks, 50
modules
client modules
Canvas modules, 259–261
discussed, 258–259
functions, list of, 269–270
messages module, 266–268
mission module, 261–262
missiondownload module, 262–266
ClientConnection, 250–256
control/client/client.cs, 160–164
control/client/interfaces/menuscreen.gui, 162
control/client/interfaces/playerinterface.gui, 165–168
control/client/misc/presetkeys.cs, 171–174
control/client/misc/screens.cs, 169–171
control/client.cs, 144–148
control/main.cs, 159
control/player.cs, 151–153
control/server/misc/item.cs, 197–202
control/server/players/player.cs, 180–186
control/server/server.cs, 175–180
control/server/weapons/crossbow.cs, 190–197
control/server/weapons/weapon.cs, 186–190
control/server.cs, 149–151
defined, 132
main.cs, 139–140
missiondownload, 246–250
server modules
discussed, 240–241
functions, list of, 270–272
Game module, 256–258
message module, 241–242
missionload, 242–246
Monster Studios Web site, 746
mood, textures, 351
mount nodes, vehicle models, 475–476
mount points, 465
mountObject method, 614
mountPose method, 611
mouse
keyboard operations and, 591
shift-clicking, 346
Move button (MilkShape Model tab), 399
movement
basic functions, 175
programmed, example of, 107–111
simple direct movement example, 105–106
MoveShape() function, 109, 117
moving structures, 606
mPow() function, 703
mRadToDeg() function, 703
msg() function, 704
mSin() function, 704
mSolveCubic() function, 704
mSolveQuadratic() function, 704
mSqrt() function, 705
mTan() function, 705
mtype parameter, 228
multiplication (*) operator, 62
music, as story line mood, 21
Mute tool (Audacity tool), 544
Myrmidon, 5

N

name spaces, 126–127
nameTolD() function, 705
navigating between bookmarks, 42
navigation keys, Emaga4, 154
nCycleExec function, 656
neck and head, player skin example, 322–326
NeHe Productions, 746
networking
direct messaging
CommandToClient function, 207–209
CommandToServer function, 206–207
discussed, 205
online cheating, fighting against, 209
GameConnection object messages, 211–217
servers, dedicated, 230–232
servers, finding
code module, 225–229
InitializeClient function, 217–218
interface module, 218–223
triggers, 209–211

networking design, Torque Game Engine, 27–28
New Audio Track command (Audacity Project menu), 548
New button (MilkShape Materials tab), 402
New command (File menu)
Audacity tool, 546
MilkShape, 392
New Image dialog box (Paint Shop Pro), 279
new keyword, 54, 735
New Label Track command (Audacity Project menu), 548
New Mission command (Mission Editor File menu), 588
New Model command (UVMapper File menu), 407
New UV Map command (Edit menu), 387, 408
newline character (n), 124
NeXe Web site, 746
nextResolution() function, 705
nextToken() function, 706
Nintendo GameCube, 8
node/bone animation, 27
nodes, scene graphs, 103
Noise Removal command (Audacity File menu), 549
NoMip flag (special materials, MilkShape), 462
Nonplayer Characters (NPCs), 322
noRenderBans property, 520
normals, gouraud shading, 99–100
Note tool (Audacity tool), 544
nothing parameter, 229
NPCs (Nonplayer Characters), 322
null strings, 255
number comparisons, 61
number token, 737

O

OBJ export options values, UVMapper tool, 388
object-oriented programming (OOP), 47
objects
3D
coordinate systems, 90–91
overview, 89
shapes, 94–95
transformation, 95–98
ActionMap, 174
controlling, 126
creating new, 125
defined, 47
flipping, 393
functions, 126
GameConnection, 211–212
initialControlSet() function, 215
onConnect() function, 213
onConnectionAccepted() function, 213
onConnectionDropped() function, 214
onConnectionError() function, 214
onConnectionRequest() function, 212
onConnectionTimedOut() function, 214
onConnectRequestRejected() function, 213–214
onConnectRequestTimedOut() function, 213
onDataBlockObjectReceived() function, 216
onDataBlocksDone() function, 216
onDrop() function, 215
onFileChunkReceived() function, 216
onGhostAlwaysObjectReceived() function, 216
onGhostAlwaysStarted() function, 217
setLagIcon() function, 215
geometric center of, 90
handles, 125–126
instantiation, 125
InteriorInstance, 125
MessageDialog, 174
method of, 126
mirroring, 393
object space, 90
rotating, 606
scaling, 606
shapes, as dynamic objects, 104
simulation, 24
offset cylindrical cap mapping option, 412
offset spherical unwrapping option, 413
OffsetSpeed property, 532
onChatMessage function, 268
onConnect() function, 213
onConnectionAccepted() function, 213
onConnectionDropped() function, 214
onConnectionError() function, 214
onConnectionRequest() function, 212
onConnectionTimedOut() function, 214
onConnectRequestRejected() function, 213–214
onConnectRequestTimedOut() function, 213
onCyclePauseEnd function, 656
onDataBlockObjectReceived() function, 216, 265

onDataBlocksDone() function, 216
onDrop() function, 215
onEnterTrigger event, 621
OnEscape method, 655
OnExit() function, 140
onFileChunkReceived() function, 216
onGhostAlwaysObjectReceived() function, 216
onGhostAlwaysStarted() function, 217
onLeaveTrigger event, 621
online cheating, fighting against, 209
OnMissionDownloadPhase function, 265
OnMissionEnded function, 258
OnMissionLoaded function, 179, 258
onMount method, 614
OnServerCreated function, 178, 258, 595, 620
OnServerDestroyed function, 258
onServerQueryStatus method, 650
OnStart() function, 140, 231
onStart() function, 159
onTickTrigger event, 621
OnWake method, 642
OOP (object-oriented programming), 47
Opacity controls (Paint Brush tool), 295
Open command (File menu)
Audacity tool, 546
MilkShape, 392
open files, Find in Files feature, 38
Open method, 654
Open Mission command (Mission Editor File menu), 588
OpenAL open-source audio API, 550
OpenALInitDriver() function, 706
OpenALShutdownDriver() function, 706
OpenGL, advantages/disadvantages, 10
operands, 72
Operate On Selected Joints Only command (MilkShape Animate menu), 394
operating systems, 8–11
operators
AND, 73
arithmetic, 63
decrement, 62
increment, 62
list of, 60–61
postdecrement, 62
postincrement, 62
relational, 72
option flags (MilkShape special materials), 461–462
order of evaluation, 62
orientAdvances property, 600
orientation planar mapping option, 411
orientOnVelocity property, 600
orientParticles property, 600
Other Settings option (Torque Game Engine (DTS) Exporter dialog box), 460
OuterChatFrame control, 639
overrideAdvances property, 600
Oxford Dynamics Web site, 746

P

package keyword, 736
packages, 132
packet loss, 28
Paint Brush tool (Paint Shop Pro), 294–296
Paint Material function, 537
Paint Shop Pro
Add Noise dialog box, 286
Airbrush tool, 296–297, 517
alpha channels, 288
Auto Proof button, 280
bitmap *versus* vector images, 288–289
Clone Brush tool, 297–298
Color dialog box, 280
Crop tool, 356
Eraser tool, 298
file types, 285–288
Fill tool, 516–517
Freehand Selection tool, 300
installing, 279
Layer palette, 291–294
Materials palette, 290–291
New Image dialog box, 279
Paint Brush tool, 294–296
Selection tool, 299
sidewalk texture example, 281–283
texture files, saving, 284–285
wooden texture example, 279–281
PanoramaScreenShot() function, 706
parameters
buddycount, 229
buddylist, 229
filterflags, 229
flags, 228
functions, 71
functions without, 70
gtype, 228

maxbots, 228
maxplayers, 228
mincpu, 228
minplayers, 228
mtype, 228
nothing, 229
passing, 71
ping, 228
port, 228
region, 228
parent controls, 346
ParseArgs function, 138, 230
ParticleData property, 601
ParticleEmitterData properties, 600
ParticleNodeData property, 599
particles
emitters, 604
examples of, 594
freestanding, 595
ParticleData property, 601
ParticleEmitterData properties, 600
ParticleEmitterNode properties, 599
particles property, 600
passing parameters, 71
Paste command
Edit menu, 588
File menu, 547
Paste Keyframes command (MilkShape Animate menu), 394
pathOnMissionLoadDone() function, 707
patterned textures, 362
patterns, search capabilities, 36
PDAs (Personal Digital Assistants), 8
pebbled textures, 361
percent sign (%), 54
percentage property, 532
periodVarianceMS property, 600
PermDisableMouse() function, 707
Personal Digital Assistants (PDAs), 8
perspective, skyboxes, 518
Phaser command (Audacity File menu), 549
phiReferenceVel property, 600
phiVariance property, 600
phong maps, 100
photography
color matching, 354–355
lighting, 355
postprocessing, 353–354
PhysicalZone class, 732
pictoral skyboxes, 514
ping parameter, 228
pipe operator (|), 61
pixel shaders, 101
Planar unwrapping method, 409, 411
PlanetTribes Web site, 742
plastic textures, 362–363
platforms, 8–11
cross-platform software, disadvantages, 11
platform layer, Torque Game Engine, 23
Play button (Audacity tool), 541
play testing, 661
Play tool (Audacity toolbar), 543
playback controls (Keyframer tool), 404
PlayDemo() function, 707
Player class, 732
player event control triggers, 210–211
player skin example
clothing, 329–332
hair texture, 327–328
hands, 329
head and neck, 322–326
PlayerKeymap, 174
PlayerSpawns property, 610
PlayJournal() function, 707
PlayStation, 8
Plot Spectrum command (Audacity View menu), 547
plug-ins, MilkShape 3D, 395–396
Plug-Ins command (MilkShape Tools menu), 395
plus sign (+), 56
PNG (Portable Network Graphics), 285–288
Point Size option (Preferences dialog box), 405
PolyCount Web site, 747
polygons
discussed, 94
incremental polygon construction, 415–416
port parameter, 228
Portable Network Graphics (PNG), 285–288
Portal brush (Torque Map Editor, QuArK), 510
portals, 509
PortInit function, 240
position property, 519
postdecrement operators, 62
postincrement operators, 62
postprocessing, photography, 353–354
precedence, evaluation, 56

Precipitation class, 732
predication strategies, 28
Prefabland Web site, 747
Preferences command (MilkShape File menu), 392
Preferences dialog box (MilkShape 3D), 404–406
presets, 159
PrevResolution() function, 707
primitive shapes, 92, 385, 415
problem decomposition, 66
problem solving techniques, 83–87
Profile class, 25
profile property, 340
ProfilerDump() function, 708
ProfilerDumpToFile() function, 708
ProfilerEnable() function, 708
ProfilerMarkerEnable() function, 708
profiles
creating and programming sound, 550–555
defined, 163
sizing properties, 164
program setup, UltraEdit-32, 32
Programmers Haven Web site, 747
programming editing tools, 751
programming editors, 31
programming languages
defined, 45
high-level languages, 47
programs. *See also* scripts
animation example, 115–117
arrays, 56–58
audio example, 119–121
campfire effects, 595–598
ChatBox interface, 637–639, 650–652
client modules
Canvas module, 259–260
messages module, 266–267
mission module, 261–262
missiondownload module, 263–265
ClientConnection module, 251–255
control/client/client.cs modules, 160–164
control/client/interfaces/playerinterface.gui module, 165–168
control/client/interfaces/splashscreen.gui module, 169
control/client/misc/presetkeys.cs module, 171–174
control/client/misc/screens.cs module, 169–171
control/client.cs module, 144–148
control/main.cs module, 159
control/player.cs module, 152–153
control/server/misc/item.cs, 197–201
control/server/players/player.cs, 180–183
control/server/server.cs, 175–178
control/server/weapons/crossbow.cs, 190–197
control/server/weapons/weapon.cs, 186–188
control/server.cs module, 149–151
debugging and problem solving techniques, 83–86
FindServer interface, 648–650
function examples, 67–69
Hello Word example, 49–51
if-else statements, 77–78
initialization, 235–240
initialization functions, 141–143
koob utility, 555–558
main.cs module, 139–140
MessageBox interface, 640–641, 653–654
missiondownload module, 247–249
missload module, 242–245
movement, 107–108
nested if statements, 79–80
rain effects, 531–532
root main modules, 134–137
rotation example, 111–113
scaling example, 113–114
servers, finding
code module, 225–229
interface module, 219–224
sound
environmental, 578–579
footstep sounds, 561–563
interface, 578–579
musical, 581–582
utterances, 563–564, 566
vehicle sounds, 573–578
weapon sounds, 566–568
storm effects, 524–528
switch statements, 81–82
water block effects, 533–534
waterfall effects, 602–604
project files, Find in Files feature, 38
Project menu (Audacity tool), 545, 548
Project Setup dialog box, 35
projects, setting up, UltraEdit-32, 32
properties
altCommand, 642
animateTexture, 601
animTexName, 601

AudioDescription, 561
AudioProfile, 561
bitmap, 164
bottom, 164
buttonType, 164
center, 164
childMargin, 343
clipColumnText, 344
colors, 601
columns, 344
command, 164, 340, 634
constantAcceleration, 601
constantThumbHeight, 225, 343
datablock, 530, 599
defaultLineHeight, 343
DepthGradient, 535
description, 552
DistortGridScale, 535
DistortMag/DistortTime, 535
dragCoefficient, 601
ejectionPeriodMS, 600
ejectionVelocity, 600
Emaga4, 154
Emaga5, 184
emitter, 599
enumerate, 344
envMapIntensity, 535
envMapOverTexture, 535
envMapUnderTexture, 535
escapeCommand, 642, 655
extent, 164, 338, 652
fadeColor, 530
fitParentWidth, 344
FlowAngle/FlowRate, 535
gravityCoefficient, 601
GuiContentProfile, 163
hardImpactSound, 576
height, 164
horizSizing, 164
hScrollBar, 225, 343
inheritedValFactor, 601
isLooping, 551
left, 164
lifetimeMS, 600–601
lifetimeVarianceMS, 600–601
MaterialList, 519
maxLength, 340
maxVelocity, 532
MinAlpha/MaxAlpha, 535
minExtent, 338
minVelocity, 532
noRenderBans, 520
OffsetSpeed, 532
orientOnVelocity, 600
orientParticles, 600
overrideAdvances, 600
ParticleData, 601
ParticleEmitterData, 600
ParticleEmitterNode, 599
ParticleNodeData, 599
particles, 600
percentage, 532
periodVarianceMS, 600
phiReferenceVel, 600
phiVariance, 600
PlayerSpawns, 610
position, 519
profile, 340
relative, 164
resizeCell, 344
right, 164
rotation, 520
scale, 519
ShoreDepth, 535
ShoreTexture, 535
sinkAllKeyEvents, 345
sizes, 601
sizeX, 531
sizeY, 531
sizing, 164
softImpactSound, 576
soundButtonDown, 580
soundButtonOver, 580
spinRandomMax, 601
spinRandomMin, 601
spinSpeed, 601
surfaceOpacity, 535
SurfaceParallex, 535
surfaceTexture, 535
TessSurface/TessShore, 535
textureName, 601
thetaMax, 600
thetaMin, 600
timeMultiple, 599
times, 601
top, 164

type, 551
UseDeptMask, 535
useEmitterColors, 600
useEmitterSizes, 600
useInvAlpha, 601
variable, 341, 344
velocity, 599
vertSizing, 164
visible, 339
visibleDistance, 520
volume, 551
vScrollBar, 225, 343
WheeledVehicleData, 618–619
width, 164
willFirstRespond, 225
windCoefficient, 601
windEffectPrecipitation, 521
windVelocity, 521
Property Selector option (Preferences dialog box), 405
pseudo-handler, 268
Psionic 3D Design Web site, 747
PurgeResources() function, 708
push buttons, GUI, 335
Push method, 171
puzzle games, 5–6

Q

Quake 3, 17
Quake III Arena command (MilkShape Tools menu), 395
QuArK
configuration, 500–501
installing, 500
map2dif reference, 767–768
Torque map, 501–502
Torque Map Editor
Cube brush, 505, 508
Portal brush, 510
Roadbed brush, 506
Subtraction brush, 509
Torque settings for, 503
Query method, 650
QueryLANServers parameters, 228
QueryMasterServer() function, 709
QueryStatus function, 225
Quick Mix command (Audacity Project menu), 548
Quick Record macro, UltraEdit-32, 43
Quit() function, 709

R

radio buttons, GUI, 335, 341
rain effects, 531–532
raster images, 288–289
raster layers, 291
ray-casting, 98
readability, improving, 86
Real-Time Strategy (RTS) genre, 7
Realm Wars Development Web site, 742
Record button (Audacity tool), 540–541
recording
Quick Record macro, 43
Standard macros, UltraEdit-32, 43–44
recoverDelay property, 184
recoverRunForceScale property, 184
rectangular coordinates, 95
Red Hat Linux distribution, 11
RedbookClose() function, 710
RedbookGetDeviceCount() function, 710
RedbookGetDeviceName() function, 710
RedbookGetLastError() function, 710
RedbookGetTrackCount() function, 710
RedbookGetVolume() function, 711
RedbookOpen() function, 711
RedbookPlay() function, 711
RedbookSetVolume() function, 711
RedbookStop() function, 711
Redo command (Edit menu)
Audacity tool, 547
MilkShape, 392
Mission Editor, 588
Redraw All Viewports button (MilkShape Model tab), 400
ReferenceDistance function, 564
reflective textures, 362
region parameter, 228
regression, testing, 660
Regroup button (MilkShape Groups tab), 401
regular expressions, 39
relational operators, 72
relative property, 164
Relight Scene command (Mission Editor Edit menu), 588
Remove All Keyframes command (MilkShape Animate menu), 394

Remove Track(s) command (Audacity Project menu), 548
RemoveField() function, 712
RemoveRecord() function, 712
RemoveTaggedString() function, 712
RemoveWork() function, 712
removing bookmarks, 42
Rename Button & Box button (MilkShape), 402–403
Rename button (MilkShape Groups tab), 401
renderFirstPerson property, 154, 184
rendering
bump mapping, 101–102
environment mapping, 102
flat shading, 99
gourand shading, 99–100
lambert shading, 99
mipmapping, 102
overview, 98
phong shading, 100
ray-casting, 98
shader programs, 101
texture mapping, 101
textured polygon, 16
repairRate property, 184
Replace dialog box, 37
requirements specification, game design, 584–585
Reset Transforms command (World menu), 590
ResetLighting() function, 712
ResetMission function, 246, 258
ResetServer function, 241
resizeCell property, 344
resource manager, Torque Game Engine, 25
RestWords() function, 713
retail games, 2
retail tools, 752–754
return character (\r), 124
return keyword, 54, 736
return statement, 71
return values, functions without, 70–71
Reverse command (Audacity File menu), 549
Reverse Vertex Order command (MilkShape Face menu), 394
rigging, character animation, 444
right-handed coordinate system, 90
right property, 164
right side view, MilkShape 3D, 382
Rigid Multifractal function, 537
Roadbed brush (Torque Map Editor, QuArK), 506
rock textures, 276
rocks, creating, 481–482
role-playing games (RPGs), 4
roll-pitch-yaw approach, rotation, 96
root animation sequence, torque-supported, 444
root main modules, 129–130, 134–137
Rotate button (MilkShape Model tab), 399
Rotate command (UVMapper Edit menu), 408
rotation, 606
images, 304–305
program example, 111–113
roll-pitch-yaw approach, 96
rotation cylindrical mapping option, 412
rotation property, 520
rotation spherical mapping option, 413
setTransform() method, 111
rough textures, 361
RPGs (role-playing games), 4–5
Rtrim() function, 713
RTS (Real-Time Strategy) genre, 7
run animation sequence, torque-supported, 444, 452–454
runEnergyDrain property, 184
runForce property, 184
runSurfaceAngle property, 154, 184

S

sample rates (Audacity tool), 544
Save As command (MilkShape File menu), 392
Save As File command (File menu), 507
Save command (MilkShape File menu), 392
Save Mission As command (Mission Editor File menu), 588
Save Mission command (Mission Editor File menu), 588
Save Model command (File menu), 387, 407
Save Project As command (Audacity File menu), 546
Save Project command (Audacity File menu), 546
Save Texture Map command (UVMapper File menu), 407
SaveJournal() function, 713
saving
layers, 294
texture files, 284–285
Scale button (MilkShape Model tab), 399
scale property, 519

scale result box mapping option, 411
scale result cylindrical cap mapping option, 412
scale result cylindrical mapping option, 412
scale result planar mapping option, 411
scale result spherical mapping option, 413
Scale tool (MilkShape), 493
scales, bump mapping, 101
scaling
images, 303–304
objects, 606
overview, 96
program example, 113–114
textures and, 358–359
scene graphs, 103–104
SceneLightingComplete function, 265
SceneObject class, 732
Schedule() function, 115, 178–179, 713
schedule method, 562
scope prefixes, 48
scoring
CheckProgress method, 624
coins, 625–627
deaths, tracking, 628–629
DoScore method, 624
laps and checkpoints, 622–625
scoutroot animation sequence, torque-supported, 444
ScreenShot() function, 713
scripts. *See also* programs
TGE (Torque Game Engine) language
overview, 123, 134
strings, 124
Torque script code fragment example, 17–19
scroll bar widgets, 336
scroll properties, 224–225
scrollbars, 342–343
SDTS (Special Data Transfer Standard), 367
seamless textures, 397
search capabilities
UltraEdit-32, 35–37
Unix-stytle syntax, 40–41
SelAssigned button (MilkShape Joints tab), 403
Select All command (Edit menu)
MilkShape, 392
Mission Editor, 588
Select button (MilkShape), 399, 401
Select By command (UVMapper Edit menu), 408
Select command (UVMapper Edit menu), 408
Select function, 537
Select Invert command (MilkShape Edit menu), 392
Select None command (Edit menu)
MilkShape, 392
Mission Editor, 588
Selection tool
Audacity toolbar, 543
Paint Shop Pro, 299
semicolon (;), 52
SendMacro() function, 206
seqmented-mesh models, 417
sequence materials, animation, 457–458
sequences, animation, 463
server control modules
control/server/misc/item.cs, 197–202
control/server/players/player.cs, 180–186
control/server/server.cs, 175–180
control/server/weapons/crossbow.cs, 190–197
control/server/weapons/weapon.cs, 186–190
server modules
discussed, 240–241
functions, list of, 270–272
Game module, 256–258
message module, 241–242
missiondownload module, 246–250
missionload, 242–246
server *versus* client design issues, 132–133
ServerMessage function, 268
serverPlay3D function, 560, 562
servers
dedicated, 230–232, 662
finding
code module, 225–229
InitializeClient function, 217–218
interface module, 218–223
hosted, 661–662
master, 217
ServerScreen class, 642
Set Empty function, 537
Set Height function, 537
Set Keyframe command (MilkShape Animate menu), 394
setActionThread method, 615
SetDefaultFov() function, 714
SetDisplayDevice() function, 714
SetEchoFileLoads() function, 714
SetField() function, 714

SetFov() function, 715
SetFSAA() function, 715
SetImageTrigger method, 629
SetInteriorFocusedDebug() function, 715
SetInteriorRenderMode() function, 715
setLagIcon() function, 215
SetLogMode() function, 715
SetModPaths() function, 716
SetNetPort() function, 716
SetNPatch() function, 716
SetOpenGLAnisotropy() function, 716
SetOpenGLInteriorMipReduction() function, 716
SetOpenGLMipReduction() function, 717
SetOpenGLSkyMipReduction() function, 717
SetOpenGLTextureCompressionHint() function, 717
SetRandomSeed() function, 717
SetRecord() function, 717
SetResolution() function, 718
SetScreenMode() function, 718
SetServerInfo() function, 718
SetShadowDetailLevel() function, 718
Settings command (UVMapper Edit menu), 408
setTransform() method, 111
SetVerticalSync() function, 718
SetWord() function, 719
SetZoomSpeed() function, 719
shade maps, 100
shader programs
  pixel shaders, 101
  vertex shaders, 101
shading
  flat, 99, 469
  gourand, 99–100
  high-contrasting, 351
  lambert, 99
  phong, 100
  smooth, 469
Shape controls (Paint Brush tool), 295
shape primitives, character models, 415
ShapeBase class, 732
ShapeBaseData class, 734
ShapeBaseImageData function, 570
shapeFile property, 154, 184
shapes
  backfaces, 95
  cylinders, MilkShape 3D, 384–386
  as dynamic objects, 104
  edges, 94
  mesh, 94
  MoveShape() function, 109, 117
  polygons, 94
  primitives, 92, 385
  sphere model example, 94
  surfaces, 94
  TestShape() function, 109
  topographical shape mapping, 417
  Torque Game Engine, 27
shareware and freeware tools, 750–752
shift-clicking, 346, 591
ShoreDepth property, 535
ShoreTexture property, 535
shortcut keys, Audacity menus, 550
Show Keyframer command (MilkShape Window menu), 397
Show Message Window command (MilkShape Window menu), 397
Show Model command (MilkShape Tools menu), 395
Show Selection command (World menu), 590
Show Viewport Caption command (MilkShape Window menu), 397
ShowMenuScreen() function, 161
side animation sequence, torque-supported, 444
sidewalk textures, 281–283
Silence command (Audacity File menu), 547
***SimCity* series,** 8
simple direct movement, translation, 105–106
SimpleNetObject class, 734
simulation
  latency problems, 28
  Torque Game Engine, 24
simulator games
  overview, 6
  strategic simulations, 8
single-quoted strings, 124
sinkAllKeyEvents property, 345
Sinus function, 537
sites. *See* Web sites
sitting animation sequence, torque-supported, 445
size
  of images, changing, 305
  terrains, 377
Size controls (Paint Brush tool), 295
sizes property, 601
sizeX property, 531

sizeY property, 531
sizing properties, 164
skeletal animation
bone movement, 446
death animation, 455–457
head attachments, 447–448, 454–455
hero rigging, 451
idle animation, 451–452
look animation, 455
run animation, 452–454
sequence materials, 457–458
torso attachment, 448–450
sketches, vehicle models, 466–467
skins. *See also* textures
character modeling, 438–443
creation process, 310–311
discussed, 20
gun creation, 494–495
player skin example
clothing, 329–332
hair texture, 327–328
hands, 329
head and neck, 322–326
soup can example, 311–315
UV unwrapping, 309–310
vehicle example, 316–321, 476
Sky class, 734
sky textures, 276–277
skyboxes
cloud layers, 521–523
discussed, 513
distorted images, 515
exploded, 514
fog, 523
images, creating, 516–518
perspective, adjusting, 518
pictoral, 514
storms
lightning, 529–531
materials, 528–529
perfect storms, 532–533
rain effects, 531–532
sound effects, 524–528
Slackware Linux distribution, 11
sliders, GUI, 335
Slowest to Fastest command (Mission Editor Camera menu), 589
smoke effects, campfire effects, 595–598
Smooth All command (MilkShape Face menu), 394
Smooth function, 537
smooth shading, 469
smooth textures, 361
Smooth Water function, 537
Smoothing function, 537
Smoothing Group Auto Smooth button (MilkShape Groups tab), 401
Smoothing Group Clear All button (MilkShape Groups tab), 401
Smoothing Group Numbers button (MilkShape Groups tab), 401
Smoothing Groups Assign button (MilkShape Groups tab), 401
Smoothing Groups Select button (MilkShape Groups tab), 401
Snap To Grid command (MilkShape Vertex menu), 393
Snap Together command (MilkShape Vertex menu), 393
Soft Focus dialog box, 314
SoftImpactSound command, 619
softImpactSound property, 576
SoftImpactSpeed command, 618
software, cross-platform, 11
solid trees, creating, 485–488
Solo tool (Audacity tool), 544
SoloPlay interface, 743–735, 743
sound. *See also* audio
Audacity tool
commands, shortcut keys to, 550
Edit menu, 545, 547
Effect menu, 545, 549
File menu, 545–546
installing, 540
main screen, 542–543
Play button, 541
Project menu, 545, 548
Record button, 540–541
sample rates, 544
toolbar tools, 543
Track Panel tools, 544
Track Types tool, 544
View menu, 545, 547
volume control, 541
client-only sounds, 560
datablocks and profiles, 550–555
discussed, 20

environmental, 578–579
footsteps, 560–563
gunshot sound-effect waveform, 21
hardImpactSound property, 576
idle engine, 576
interface, 578–579
koob utility, 555–558
musical, 580–582
OpenAL open-source audio, 550
softImpactSound property, 576
soundButtonDown property, 580
soundButtonOver property, 580
storm effects, 524–528
utterances, 563–565
vehicle sounds, 572–578
weapon sounds, 565–572
world sounds, 559
SourceForge.net Web site, 747
SpamAlert function, 242
SpamMessageTimeout method, 242
spamming, 242
spawn point system, 609
SpawnPlayer method, 151
special characters, in find function, 37
Special Data Transfer Standard (SDTS), 367
special materials (MilkShape), 460–463
Specular & Specular Slider button (MilkShape Materials tab), 402
speedDamageScale property, 184
Sphere button (MilkShape Model tab), 399
sphere model example, 94
Sphere tool (MilkShape), 481
Spherical unwrapping method, 413
spinRandomMax property, 601
spinRandomMin property, 601
spinSpeed property, 601
Split command (Audacity File menu), 547
split planar mapping option, 411
spread facets at poles cap mapping option, 412
spread facets at poles spherical mapping option, 413
spread of terrains, 366
SquealSound command, 619
Standard macro, UltraEdit-32, 43–44
standard strings, 124
StartGame function, 178
StartHeartbeat() function, 719
StartRecording() function, 719
state handlers, 570
state machine, 565
statements
assigned, 52
compound, 52
defined, 52
if
nested if statements, 79–80
overview, 75–76
if-else, 76–79
return, 71
switch, 81–82
StaticShape class, 735
Statistics command (UVMapper Help menu), 408
Step control (Paint Brush tool), 295
stone textures, 275, 359
Stop tool (Audacity tool), 543
StopHeartbeat() function, 719
StopRecording() function, 720
StopServerQuery() function, 720
storms
lightning, 529–531
materials, 528–529
perfect storm effects, 532–533
rain effects, 531–532
sound effects, 524–528
straight lines, creating, 294
strategy games, 7–8
Strchr() function, 720
Strcmp() function, 720–721
string constant token, 737
strings
assignment operators, 59
concatenation operators, 59
defined, 59
double-quoted, 124
null, 255
single-quoted, 124
standard, 124
tagged, 60, 124
Stripchars() function, 721
StripMLControlChars() function, 721
StripTrailingSpaces() function, 721
Strlen() function, 721
Strlwr() function, 722
Strpos() function, 722
Strreplace() function, 722
Strstr() function, 722

StrToPlayerName() function, 723
structures
bridges, 505–508
houses, 508–511
interior, 499
moving, 606
Strupr() function, 723
stub routines, 138
Style button (Materials palette), 291
Sub flag (special materials, MilkShape), 462
Subdivide 3 command (MilkShape Face menu), 394
Subdivide 4 command (MilkShape Face menu), 394
subscripts, arrays and, 59
substructures, 275–276
subtraction (-) operator, 61
Subtraction brush (Torque Map Editor, QuArK), 509
subtree control, 129, 133
superclasses, 129
support infrastructure
administrative tools, 22
auto-update programs, 22
bulletin boards, 22
databases, 22
forums, 22
Web sites, 21
surfaceOpacity property, 535
SurfaceParallax property, 535
surfaces, 94
surfaceTexture property, 535
SuSe Linux distribution, 11
switch keyword, 54, 736
switch statement, 81–82
SwitchBitDepth() function, 723
syntax highlighting, 31

T

tab character (\t), 124
tabs
Groups (MilkShape toolbox), 400
Joints (MilkShape 3D), 403
Materials (MilkShape toolbox), 400, 402
Misc (Preferences dialog box), 404
Model (MilkShape toolbox), 398–400
Viewport (Preferences dialog box), 404
tagged strings, 60, 124
tags, 133
TCPObject class, 735
technological textures, 277
TellAll() function, 207
TelnetSetParameters() function, 723
Terraformer class, 735
Terrain Editor, 534–537, 590–592
Terrain Editor Settings command (Mission Editor Edit menu), 588
Terrain File function, 537
Terrain Terraform Editor, 592
Terrain Texture Editor, 592–593
terrains, 20
blur effects, 374
covers, 369, 378–380
creating, 371–377
DEM (Digital Elevation Model), 367
external method approach to, 367
fidelity, 366
freedom of, 366
height-maps, 367–368
internal method approach to, 368
mirroring, 594
SDTS (Spatial Data Transfer Standard), 367
sizes, 377
spread of, 366
Terrain Manager, 368
TessSurface/TessShore property, 535
test editors, 31
testing
alpha test phase, 661
beta test phase, 661
gun creation, 495–496
methodologies, 660
play testing, 661
regression, 660
rock creation, 483
test harnesses, 661
tree creation
billboard trees, 489–490
solid trees, 487–488
vehicle models, 477–478
TestShape() function, 109
text
anti-aliasing, 306
fonts, 306

kerning, 306
leading, 306
Texture Browser Button button (MilkShape Materials tab), 402
Texture button (Materials palette), 291
Texture Coordinate Editor command (MilkShape Window menu), 397
Texture Coordinate Editor dialog box, 494–495
Texture Coordinate Editor (MilkShape), 406
texture mapping, Texture Coordinate Editor feature, 406
textured polygon rendering, 16
textureName property, 601
textures. *See also* skins
brick, 359
cloud, 522–523
discussed, 20
distant objects, 278
fabric, 362
files, saving, 284–285
glass, 278
irregular, 360
metal, 277
metallic, 362
as mood alteration, 351
patterned, 362
pebbled, 361
plastic, 362–363
reflective, 362
rock, 276
rough, 361
scaling issues, 358–359
seamless, 397
sidewalk, 281–283
sky, 276–277
smooth, 361
sources for
artwork, 357–358
photography, 352
stone, 275, 359
substructures, 275–276
technological, 277
terrain accents, 277
texture mapping, 275
vehicle, 278
water, 276
wooden, 275, 279–281, 361
***The Incredible Machine* series,** 6
Thermal Erosion function, 537
thetaMax property, 600
thetaMin property, 600
ThinkTanks, 3
Third-Person Point-of-View (3rd PPOV) games, 3
thumb widgets, 336
thunder, lightning effects, 530
tilde (~), 39, 231
Tile command (UVMapper Edit menu), 408
tiles, terrains, 365–366
tiling
images, 359–360
terrains, 369, 371
Time Shift tool (Audacity toolbar), 543
timeMultiple property, 599
times property, 601
TireEmitter command, 618
Toggle Camera command (Mission Editor Camera menu), 589
Toggle3rdPPOVLook function, 175
ToggleFullScreen() function, 723
ToggleInputState() function, 724
ToggleMessageBox function, 637
ToggleNPatch() function, 724
ToggleState method, 655
tokens, 737
Tool Bar, Torque GUI Editor, 347
toolbox, MilkShape
Groups tab, 400–401
Joints tab, 403
Keyframer tool, 403–404
Materials tab, 400, 402
Model tab, 398–400
tools
assemblers, 46
as support infrastructure, 22
Tools command (UVMapper Edit menu), 408
Tools menu (Milkshape 3D), 395
top-down approach, problem decomposition, 66
top property, 164
top view, MilkShape 3D, 382
topographical shape mapping, 417
Torque
Torque Game Engine
3D world rendering system, 26
bandwidth strategies, 28
bitmap support, 25
console library, 24

control flow, 23
extrapolation strategies, 28
input model, 24
installing, 29
interior library, 27
interpolation strategies, 28
networking design, 27–28
platform layer, 23
prediction strategies, 28
resource manager, 25
script code fragment example, 17–19
shapes and animation, 27
simulation, 24
strings, 124
terrain library, 26–27
Torque Script console language, 24
utility functions, 25
Torque GUI Editor
Content Editor, 345–346
Control Inspector, 347
Control Tree, 346
creating interfaces using, 348–349
keyboard commands, 348
launching, 345
Tool Bar, 347
Torque Map Editor (QuArK)
Cube brush, 505
Portal brush, 510
Roadbed brush, 506
Subtraction brush, 509
Torque-related Web sites, 741–742
torso
character models, 423–430
skeletal animation, 448–450
total frames box (Keyframer tool), 404
totalizers, 63
Trace() function, 141, 724
Track Delete tool (Audacity tool), 544
Track menu (Audacity tool), 544
Track Panel tools (Audacity tool), 544
Track Types (Audacity tool), 544
transformation
full, 97
getTransform() method, 109
overview, 95
rotation, 96
scaling, 96
translation, 97
translations
overview, 97
programmed movement, 107–111
simple direct movement, 105–106
transparency, 289
Transparency Slider button (MilkShape Materials tab), 402
Transparent button (Materials palette), 291
trees, creating
billboard trees, 488–490
discussed, 483
solid trees, 485–488
Tribes 2, 17
Trigger class, 735
trigger events
animation, 209–210
area, 209
creating, 620–621, 623
kill tracking, 629–630
onEnterTrigger, 621
onLeaveTrigger, 621
onTickTrigger, 621
player event control, 210–211
scoring
CheckProgress method, 624
coins, 625–627
deaths, tracking, 628–629
DoScore method, 624
laps and checkpoints, 622–625
SetImageTrigger method, 629
TriggerData class, 735
Trim() function, 724
true keyword, 54, 736
Tubetti Enterprises, 4
Turbolinux distribution, 11
Turbulence function, 537
Turn Edge command (MilkShape Face menu), 394
type property, 551

U

U-V Coordinate Mapping, 309–310
UEPM (UltraEdit Project Maker), 32
UltraEdit-32
bookmark capabilities, 41
configuring, 33–35
discussed, 31
Find in Files feature, 38

grep command capabilities, 39–41
Help feature, 45
installing, 32
macro commands, 43
program setup, 32
Project Setup dialog box, 35
projects and files, setting up, 32
search capabilities, 35–37
UEPM (UltraEdit Project Maker), 32
UltraEdit Project Maker (UEPM), 32
underscore (_), 53
Undo command (Edit menu)
Audacity tool, 547
MilkShape, 392
Mission Editor, 588
Unfloat Palette command (Audacity View menu), 547
Unhide All command (MilkShape Edit menu), 392
Unix-style syntax, search capabilities, 40–41
Unlock Selection command (World menu), 590
unmounting, dismounting, 615–616
Unreal II, 17
Unreal Tournament command (MilkShape Tools menu), 395
Unweld command (MilkShape Vertex menu), 393
Unweld Radial command (MilkShape Vertex menu), 393
Update method, 650
UpdateLap method, 621
UpdateLightingProgress function, 266
updates, for loop, 66
Usage() function, 138
UseDepthMask property, 535
useEmitterColors property, 600
useEmitterSizes property, 600
useInvAlpha property, 601
user ID lists, 255
utility functions, Torque Game Engine, 25
utterances, sound effects, 563–565
UV unwrapping, 309–310, 386
UVMapper tool
discussed, 386
hot keys, 410
menus
Edit menu, 407–408
File menu, 407
Help, 407, 409
OBJ export options values, 388
overview, 406
unwrapping methods
Box method, 409, 411
Cylindrical, 409, 412
Cylindrical Cap method, 410, 412
list of, 407
Planar method, 409, 411
Spherical method, 413

V

ValidateMemory() function, 724
variable property, 341, 344
variable token, 737
variables
case-sensitivity, 54
defined, 53
identifiers, 54
keywords, 53
member, 127
vector images, 288–289
vector layers, 291
vector triplets, 97
VectorAdd() function, 725
VectorCross() function, 725
VectorDist() function, 725
VectorDot() function, 725
VectorLen() function, 725
VectorNormalize() function, 726
VectorOrthoBasis() function, 726
VectorScale() function, 726
VectorSub() function, 726
vehicle models
body of vehicle, 467–472
collision mesh, 476
collisions, 612–613
fenders, 473–474
mount nodes, 475–476
mounting, 611
sketches, 466–467
skins, 476
sounds, 572–578
testing, 477–478
textures, 278
WheeledVehicleData property, 618–619
wheels, 476–477
velocity property, 599
velocityVariance property, 600
Vertex button (MilkShape Model tab), 399
Vertex menu (MilkShape 3D), 391, 393

vertex shaders, 101
Vertex tool (MilkShape), 488
vertSizing property, 164
VideoSetGammaCorrection() function, 260, 726
View menu (Audacity tool), 545, 547
Viewport tab (Preferences dialog box), 404
Viewports command (MilkShape Window menu), 397
visibility tracks, animation, 27
visible property, 339
visibleDistance property, 520
volume control, Audacity tool, 541
volume property, 551
volumetric fog, 523
vScrollBar property, 225, 343

W

Wahwah command (Audacity File menu), 549
walk animation sequence, torque-supported, 444
Warn() function, 83, 727
warped images, 518
water block effects, 533–534
water textures, 276
WaterBlock class, 735
waterfall effects, 602–604
waveforms, sound effects, 21
WeaponImage function, 570
weapons
- control/server/weapons/crossbow.cs module, 190–197
- control/server/weapons/weapon.cs module, 186–190
- gun creation
  - model building, 490–494
  - skins, 494–495
  - testing, 495–496
- sound effects, 565–572

Web sites
- game development, 743–747
- GarageGames, 27
- Linux tool sources, 749
- Macintosh tool sources, 749
- as support infrastructure, 21
- Torque-related, 741–742

Weld Together command (MilkShape Vertex menu), 393
WheeledVehicleData properties, 618–619
WheelImpactSound command, 619
wheels, vehicle models, 476–477
while keyword, 54, 736
while loop, 64–65
white space, 86, 645
widgets, 336
width property, 164
wildcards, search capabilities, 36
willFirstRespond property, 225, 342–343
wind, cloud layers, 521
windCoefficient property, 601
windEffectPrecipitation property, 521
Window menu (MilkShape 3D), 397
Windows operating system, 8, 10
windVelocity property, 521
Winter, David A. (*Maximum Football*), 7
wooden textures, 275, 279–281, 361
working environment, MilkShape 3D, 382
World Editor Settings command (Mission Editor Edit menu), 588
World menu (World Editor), 590
world objects, 26
world sounds, 559
world space, coordinate systems, 91
world units (WU), 378
World War II Online, 22
WorldEditor class, 735
Wotsit's Format Web site, 747
WU (world units), 378

X

XYZ-axis system, 90–93

Z

z-flat shading, 99
Zoom In command (Audacity View menu), 547
Zoom Normal command (Audacity View menu), 547
zoom options, MilkShape 3D, 383
Zoom Out command (Audacity View menu), 547
Zoom tool (Audacity toolbar), 543

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

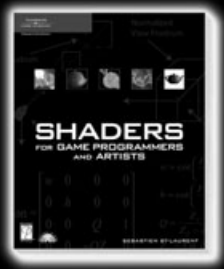
GOT GAME?

COMING SPRING 2004!

Programming
Role Playing Games
with DirectX, 2nd Edition
1-59200-315-X ■ $49.99

Beginning C++
Game Programming
1-59200-205-6 ■ $29.99

The Dark Side
of Game Texturing
1-59200-350-8 ■ $39.99



Shaders for Game
Programmers and Artists
1-59200-092-4 ■ $39.99

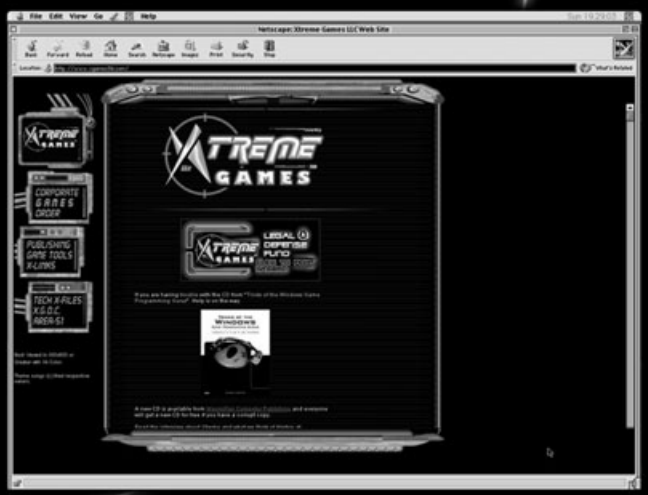
PHP
Game Programming
1-59200-153-X ■ $39.99

Premier Press™
A division of Course Technology

PREMIER PRESS
GAME DEVELOPMENT

Call **1.800.354.9706** to order
Order online at **www.courseptr.com**

TAKE YOUR GAME TO THE XTREME!



Xtreme Games LLC was founded to help small game developers around the world create and publish their games on the commercial market. Xtreme Games helps younger developers break into the field of game programming by insulating them from complex legal and business issues. Xtreme Games has hundreds of developers around the world. If you're interested in becoming one of them, then visit us at **www.xgames3d.com.**

www.xgames3d.com

XTREME GAMES

Gamedev.net
The most comprehensive game development resource
The latest news in game development
The most active forums and chatrooms anywhere, with insights and tips from experienced game developers
Links to thousands of additional game development resources
Thorough book and product reviews
Over 1000 game development articles!
Game design
Graphics
DirectX
OpenGL
AI
Art
Music
Physics
Source Code
Sound
Assembly
And More!
Gamedev.net
OpenGL is a registered trademark of Silicon Graphics, Inc.
Microsoft and DirectX are registered trademarks of Microsoft Corp. in the United States and/or other countries.

Team LRN

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:
The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:
The enclosed disc is warranted by Course PTR to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Course PTR will provide a replacement disc upon the return of a defective disc.

Limited Liability:
THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL COURSE PTR OR THE AUTHOR BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF COURSE PTR AND/OR THE AUTHOR HAS PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

Disclaimer of Warranties:
COURSE PTR AND THE AUTHOR SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

Other:
This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Course PTR regarding use of the software.

Team LRN